

GUIDE TO VAX/VMS FILE APPLICATIONS

Order Number: AI-Y508B-TE

April 1986

This document is intended for application programmers and designers of programs that use VAX RMS files.

Revision/Update Information:

This revised document supersedes the Guide to VAX/VMS File Applications Manual, Version 4.0

Software Version:

VAX/VMS Version 4.4

**digital equipment corporation
maynard, massachusetts**

April 1986

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright ©1986 by Digital Equipment Corporation
All Rights Reserved

The postpaid **READER'S COMMENTS** form on the last page of this document requests the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem-10	PDP	VT
DECSYSTEM-20	PDT	
DECUS	RSTS	
DECwriter	RSX	

digital

ZK-3068

**HOW TO ORDER ADDITIONAL DOCUMENTATION
DIRECT MAIL ORDERS**

USA & PUERTO RICO*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire
03061

CANADA

Digital Equipment
of Canada Ltd.
100 Herzberg Road
Kanata, Ontario K2K 2A6
Attn: Direct Order Desk

INTERNATIONAL

Digital Equipment Corporation
PSG Business Manager
c/o Digital's local subsidiary
or approved distributor

In Continental USA and Puerto Rico call 800-258-1710.

In New Hampshire, Alaska, and Hawaii call 603-884-6660.

In Canada call 800-267-6215.

* Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575).

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Westminister, Massachusetts 01473.

This document was prepared using an in-house documentation production system. All page composition and make-up was performed by \TeX , the typesetting system developed by Donald E. Knuth at Stanford University. \TeX is a registered trademark of the American Mathematical Society.

Contents

PREFACE	xv
---------	----

NEW AND CHANGED FEATURES	xix
--------------------------	-----

CHAPTER 1 INTRODUCTION	1-1
------------------------	-----

1.1	FILE CONCEPTS	1-3
1.1.1	Disk Structures	1-5
1.1.1.1	Logical Structures • 1-5	
1.1.1.2	Physical Structures • 1-8	
1.1.1.3	Files-11 On-Disk Structure Index File • 1-10	
1.1.2	Tape Concepts	1-11

1.2	VOLUME PROTECTION	1-13
-----	-------------------	------

1.3	RECORD MANAGEMENT SERVICES	1-14
1.3.1	File Definition Language	1-14
1.3.2	VAX RMS Data Structures	1-15
1.3.3	VAX RMS Services	1-15

1.4	VAX RMS UTILITIES	1-16
1.4.1	The Analyze/RMS_File Utility	1-16
1.4.2	The Convert Utility	1-17
1.4.3	The Convert/Reclaim Utility	1-17
1.4.4	The Create/FDL Utility	1-18
1.4.5	The Edit/FDL Utility	1-19

1.5	PROCESS AND SYSTEM RESOURCES FOR FILE APPLICATIONS	1-20
1.5.1	Memory Requirements	1-20
1.5.2	Process Asynchronous I/O Limits	1-21
1.5.3	Process Record-Locking Quota	1-22
1.5.4	Process Open File Limit	1-22

1.6	SETTING PROCESS AND SYSTEM DEFAULTS	1-22
-----	-------------------------------------	------

CHAPTER 2 CHOOSING A FILE ORGANIZATION **2-1**

2.1	RECORD CONCEPTS	2-2
2.1.1	Record Access Modes	2-2
2.1.1.1	Sequential Access • 2-4	
2.1.1.2	Random Access by Key Value or Relative Record Number • 2-7	
2.1.1.3	Random Access by Record's File Address • 2-9	
2.1.2	Record Formats	2-9
2.1.2.1	Fixed-Length Records • 2-11	
2.1.2.2	Variable-Length Records • 2-11	
2.1.2.3	Variable-Length with Fixed-Length Control Records • 2-13	
2.1.2.4	Stream Format • 2-15	
2.2	FILE ORGANIZATION CONCEPTS	2-16
2.2.1	Sequential File Organization	2-18
2.2.2	Relative File Organization	2-20
2.2.3	Indexed File Organization	2-22
2.2.3.1	Sequentially Retrieving Indexed Records • 2-23	
2.2.3.2	Index Keys • 2-24	
2.2.3.3	Other Key Characteristics • 2-24	
2.2.3.4	Specifying Sort Order • 2-26	

CHAPTER 3 PERFORMANCE CONSIDERATIONS **3-1**

3.1	DESIGN CONSIDERATIONS	3-1
3.1.1	Speed	3-1
3.1.2	Space	3-2
3.1.3	Shared Access	3-3
3.1.4	Impact on Applications Design	3-4
3.2	TUNING	3-4
3.2.1	File Design Attributes	3-4
3.2.1.1	Initial File Allocation • 3-5	
3.2.1.2	Contiguity • 3-5	
3.2.1.3	Extending a File • 3-5	
3.2.1.4	Units of Input/Output • 3-7	
3.2.1.5	Multiple Areas for Indexed Files • 3-7	
3.2.1.6	Bucket Fill Factor for Indexed Files • 3-8	

Contents

3.2.2	Processing Options	3-8
3.2.2.1	Multiple Buffers • 3-9	
3.2.3	Global Buffers	3-10
3.2.4	Deferred Write Processing	3-10
3.2.5	Read-Ahead and Write-Behind Processing	3-11
<hr/>		
3.3	TUNING A SEQUENTIAL FILE	3-11
3.3.1	Block Spanning	3-12
3.3.2	Multiblock Size	3-13
3.3.3	Number of Buffers	3-13
3.3.4	Global Buffers	3-14
3.3.5	Specifying Read-Ahead and Write-Behind	3-14
<hr/>		
3.4	TUNING A RELATIVE FILE	3-15
3.4.1	Bucket Size	3-15
3.4.2	Number of Buffers	3-16
3.4.3	Global Buffers	3-18
3.4.4	Using Deferred Write	3-18
<hr/>		
3.5	TUNING AN INDEXED FILE	3-19
3.5.1	File Structure	3-19
3.5.1.1	Prologues • 3-19	
3.5.1.2	Primary Index Structure • 3-21	
3.5.1.3	Alternate Index Structures • 3-23	
3.5.1.4	Records • 3-23	
3.5.1.5	Keys • 3-26	
3.5.1.6	Areas • 3-27	
3.5.2	Optimizing File Performance	3-29
3.5.2.1	Bucket Size • 3-29	
3.5.2.2	Fill Factor • 3-31	
3.5.2.3	Number of Buffers • 3-31	
3.5.2.4	Global Buffers • 3-33	
3.5.2.5	Using Deferred Write • 3-33	
<hr/>		
3.6	PROCESSING IN A VAXCLUSTER	3-34
3.6.1	VAXCluster Shared Access	3-35
3.6.1.1	Locking Considerations • 3-35	
3.6.1.2	I/O Considerations • 3-36	
3.6.2	Performance Recommendations	3-36

CHAPTER 4	CREATING AND POPULATING FILES	4-1
<hr/>		
4.1	FILE CREATION CHARACTERISTICS	4-1
4.1.1	Using VAX RMS Control Blocks	4-1
4.1.1.1	File Access Block • 4-2	
4.1.1.2	Extended Attribute Blocks • 4-2	
4.1.2	Using File Definition Language	4-3
4.1.2.1	Using the Edit/FDL Utility • 4-3	
4.1.2.2	Designing an FDL File • 4-15	
4.1.2.3	Setting Characteristics for FDL Files • 4-18	
4.1.3	Using the FDL Routines	4-19
<hr/>		
4.2	CREATING A FILE	4-22
4.2.1	Using the VAX RMS Create Service	4-22
4.2.2	Using the Create/FDL Utility	4-22
4.2.3	Using the Convert Utility	4-23
4.2.4	Using the FDL\$CREATE Routine	4-23
<hr/>		
4.3	DEFINING FILE PROTECTION	4-26
4.3.1	UIC-Based Protection	4-27
4.3.2	ACL-Based Protection	4-28
<hr/>		
4.4	POPULATING A FILE	4-28
4.4.1	Using the Convert Utility	4-28
4.4.2	Using the Convert Routines	4-29
<hr/>		
4.5	SUMMARY OF FILE CREATION OPTIONS	4-35
4.5.1	File Disposition Options	4-35
4.5.2	File Characteristics	4-36
4.5.3	File Allocation and Positioning	4-38

CHAPTER 5 LOCATING AND NAMING FILES **5-1**

5.1	UNDERSTANDING FILE SPECIFICATIONS	5-1
5.1.1	File Specification Formats _____	5-4
5.1.2	Using File Specification Defaults _____	5-5
5.2	LOGICAL NAMES AND PARSING	5-6
5.2.1	Example Use of Logical Names _____	5-7
5.2.2	Types of Logical Names _____	5-7
5.2.3	Introduction to File Specification Parsing _____	5-9
5.3	USING ONE FILE SPECIFICATION TO LOCATE MANY FILES	5-11
5.3.1	Processing One File _____	5-19
5.3.2	Processing Many Files _____	5-20
5.3.3	Processing One or Many Files _____	5-21

CHAPTER 6 ADVANCED USE OF FILE SPECIFICATIONS **6-1**

6.1	HOW VAX RMS APPLIES DEFAULTS	6-1
6.2	UNDERSTANDING VAX RMS PARSING	6-4
6.2.1	Checking for Open-By-Name Block _____	6-5
6.2.2	File Specification Formats and Translating Logical Names _____	6-6
6.2.3	Special Parsing Conventions _____	6-8
6.3	DIRECTORY SYNTAX CONVENTIONS AND DIRECTORY CONCATENATION	6-14
6.3.1	Using Normal Directory Syntax _____	6-14
6.3.2	Rooted Directory Syntax Applications _____	6-16
6.3.3	Using Rooted Directory Syntax _____	6-17
6.3.4	Concatenating Rooted Directory Specifications _____	6-19
6.3.5	An Example of Rooted Directory Use _____	6-22
6.4	USING PROCESS-PERMANENT FILES	6-23

CHAPTER 7 FILE SHARING AND BUFFERING **7-1**

7.1	FILE SHARING	7-1
7.1.1	Types of File Sharing and Record Streams	7-2
7.1.2	Interlocked Interprocess File Sharing	7-7
7.1.3	User-Interlocked Interprocess File Sharing	7-9
7.2	RECORD LOCKING	7-10
7.2.1	Automatic Record Locking	7-11
7.2.2	Manual Record Unlocking	7-13
7.2.3	Record Locking and Unlocking	7-13
7.2.4	Programming Techniques for Shared Files	7-17
7.2.4.1	Waiting for Records • 7-17	
7.2.4.2	Try Again on Record Lock Errors • 7-18	
7.3	LOCAL AND SHARED BUFFERING TECHNIQUES	7-19
7.3.1	Record Transfer Modes	7-19
7.3.2	Understanding Buffering	7-20
7.3.3	Buffering for Sequential Files	7-23
7.3.4	Buffering for Relative Files	7-23
7.3.5	Buffering for Indexed Files	7-24
7.3.6	Using Global Buffers for Shared Files	7-25

CHAPTER 8 RECORD PROCESSING **8-1**

8.1	RECORD OPERATIONS	8-1
8.2	PRIMARY VAX RMS SERVICES	8-1
8.2.1	Locating and Retrieving Records	8-3
8.2.2	Inserting Records	8-4
8.2.3	Updating Records	8-5
8.2.4	Deleting Records	8-6
8.3	SECONDARY SERVICES	8-7
8.4	RECORD ACCESS FOR THE VARIOUS FILE ORGANIZATIONS	8-7

Contents

8.4.1	Processing Sequential Files	8-9
8.4.1.1	Sequential Access • 8-9	
8.4.1.2	Random Access • 8-10	
8.4.2	Processing Relative Files	8-10
8.4.2.1	Sequential Access • 8-11	
8.4.2.2	Random Access • 8-11	
8.4.3	Processing Indexed Files	8-12
8.4.3.1	Sequential Access • 8-13	
8.4.3.2	Random Access • 8-14	
8.4.4	Access by Record's File Address	8-16
<hr/>		
8.5	BLOCK INPUT/OUTPUT	8-16
<hr/>		
8.6	CURRENT RECORD CONTEXT	8-18
8.6.1	Current Record	8-19
8.6.2	Next Record	8-20
<hr/>		
8.7	SYNCHRONOUS AND ASYNCHRONOUS OPERATIONS	8-21
8.7.1	Using Synchronous Operations	8-22
8.7.2	Using Asynchronous Operations	8-23
<hr/>		
CHAPTER 9 RUN-TIME OPTIONS		9-1
<hr/>		
9.1	SPECIFYING RUN-TIME OPTIONS	9-1
9.1.1	Using the FDL Editor	9-2
9.1.2	Using Language Statements and VAX RMS	9-5
<hr/>		
9.2	OPTIONS RELATED TO OPENING AND CLOSING FILES	9-6
9.2.1	File Access and Sharing Options	9-7
9.2.2	File Specifications	9-7
9.2.3	File Performance Options	9-8
9.2.3.1	Extension Size • 9-9	
9.2.3.2	Window Size • 9-9	
9.2.3.3	Summary of Performance Options • 9-10	
9.2.4	Record Access Options	9-12
9.2.5	Options for Adding Records	9-12
9.2.6	Options for Data Reliability	9-13
9.2.7	Options for File Disposition	9-14
9.2.8	Options for Indexed Files	9-15
9.2.9	Options for Magnetic Tape Processing	9-16

9.2.10	Options for Nonstandard File Processing	9-17
--------	---	------

9.3	SUMMARY OF RECORD OPERATION OPTIONS	9-18
9.3.1	Record Retrieval Options	9-18
9.3.2	Record Insertion Options	9-21
9.3.3	Record Update Options	9-24
9.3.4	Record Deletion Options	9-25
9.4	RUN-TIME EXAMPLE	9-26

CHAPTER 10	MAINTAINING FILES	10-1
------------	-------------------	------

10.1	VIEWING FILE CHARACTERISTICS	10-1
10.1.1	Performing an Error Check	10-2
10.1.2	Generating a Statistics Report	10-7
10.1.3	Using Interactive Mode	10-14
10.1.4	Examining a Sequential File	10-15
10.1.5	Examining a Relative File	10-20
10.1.6	Examining an Indexed File	10-22

10.2	GENERATING AN FDL FILE FROM A DATA FILE	10-28
------	---	-------

10.3	OPTIMIZING AND REDESIGNING FILE CHARACTERISTICS	10-31
10.3.1	Redesigning an FDL File	10-33
10.3.2	Optimizing a Data File	10-34

10.4	MAKING A FILE CONTIGUOUS	10-35
10.4.1	Using the Copy Utility	10-35
10.4.2	Using the Convert Utility	10-36
10.4.3	Reclaiming Buckets in Prolog 3 Files	10-36

10.5	REORGANIZING A FILE	10-37
------	---------------------	-------

10.6	MAKING ARCHIVE COPIES	10-37
------	-----------------------	-------

APPENDIX A EDIT/FDL OPTIMIZATION ALGORITHMS **A-1**

A.1	ALLOCATION	A-1
A.2	EXTENSION SIZE	A-1
A.3	BUCKET SIZE	A-2
A.4	GLOBAL BUFFERS	A-2
A.5	INDEX DEPTH	A-3

GLOSSARY

INDEX

EXAMPLES

4-1	Sample FDL Edit Session	4-7
4-2	Sample FDL File	4-13
4-3	Using FDL Routines in a PASCAL Program	4-20
4-4	Using the FDL\$CREATE Routine in a FORTRAN Program	4-24
4-5	Using the FDL\$CREATE Routine from a COBOL Program	4-25
4-6	Using the CONVERT Routines in a FORTRAN Program	4-30
4-7	Using the CONVERT Routines in COBOL Program	4-32
5-1	Using Logical Names for Remote File Access	5-8
5-2	VAX MACRO Routine Called by a BASIC Useropen Program	5-14
5-3	Using the Parse, Search, and Open Services	5-17
6-1	Example of Rooted Directory Syntax	6-23
7-1	Example of Waiting to Access a Locked Record	7-19
9-1	Using FDL\$PARSE and FDL\$RELEASE	9-26
10-1	Using ANALYZE/RMS_FILE to Create a Check Report	10-3
10-2	Using ANALYZE/RMS_FILE to Create a Statistics Report	10-7
10-3	Examining a Sequential File	10-18

10-4	Examining a Relative File	10-21
10-5	Examining an AREA DESCRIPTOR	10-24
10-6	Examining a Primary Record	10-27
10-7	Examining an Alternate Record	10-28
10-8	KEY and ANALYSIS_OF_KEY Sections in an FDL File	10-29

FIGURES

1-1	Files-11 On-Disk Structure Hierarchy	1-6
1-2	File Extents	1-7
1-3	Tracks and Cylinders	1-9
1-4	Interrecord Gaps	1-12
1-5	CONVERT Creates Data Files	1-18
1-6	CREATE/FDL Creates Empty Data Files	1-19
2-1	Sequential Access to a Sequential File	2-4
2-2	Sequentially Retrieving Records in a Relative File	2-5
2-3	Sequentially Storing Records in a Relative File	2-6
2-4	Random Access by Relative Record Number	2-7
2-5	Random Access by Record's File Address	2-10
2-6	Comparison of Fixed- and Variable-Length Records	2-12
2-7	Writing a Variable with Fixed-Length Control Record	2-14
2-8	Retrieving a Variable with Fixed-Length Control Record	2-15
2-9	Sequential File Organization	2-18
2-10	Relative File Organization	2-21
2-11	Variable-Length Records in Fixed-Length Cells	2-21
2-12	Single-Key Indexed File Organization	2-26
2-13	Multiple-Key Indexed File Organization	2-27
3-1	VAX RMS Index Structure	3-22
3-2	A Primary Index Structure	3-24
3-3	Finding the Record with Key 14	3-25
4-1	A Line_Plot Graph	4-16
4-2	A Surface_Plot Graph	4-17
4-3	Design Mnemonics	4-18
7-1	Shared File Access	7-1
7-2	VAX RMS Buffers and the User Program	7-21
7-3	Using Global Buffers for a Shared File	7-26
8-1	Using RFA Access to Establish Record Position	8-17

Contents

10-1	Tree Structure for Sequential Files	10-15
10-2	Record Layout and Content for SEQ.DAT	10-17
10-3	Tree Structure of Relative Files	10-20
10-4	AREA DESCRIPTOR Path	10-23
10-5	KEY DESCRIPTOR Path	10-25
10-6	Structure of Primary Records	10-26
10-7	Structure of Alternate Records	10-26
10-8	The VAX RMS Tuning Cycle	10-32

TABLES

2-1	Record Access Modes and File Organizations	2-3
2-2	File Organization Characteristics and Capabilities	2-16
2-3	Sequential File Organization Advantages and Disadvantages	2-19
2-4	Relative File Organization Advantages and Disadvantages	2-22
2-5	Indexed File Organization Advantages and Disadvantages	2-28
4-1	Summary of Edit/FDL Utility Commands	4-4
4-2	EDIT/FDL scripts	4-6
6-1	File Specification Defaults	6-1
6-2	Application of Defaults Example	6-3
7-1	File Access Record Operations	7-4
7-2	File Sharing Record Operations	7-5
7-3	Initial File Sharing and Subsequent File Access	7-7
7-4	Initial File Access and Subsequent File Sharing	7-8
8-1	Record Operations and File Organizations	8-3
8-2	Record Access Stream Context	8-18
10-1	ANALYZE/RMS_FILE Command Summary	10-14

Preface

This guide describes how to use file organizations; how to create, populate, locate, share, and maintain files; and how to process records. It concentrates on the human and program interfaces provided by the File Definition Language (FDL) and VAX Record Management Services (VAX RMS). Both these facilities can be used to define file and record characteristics, the location of a file, run-time characteristics, and related options. VAX RMS services can be used to create and access files, and to process records. Examples of using FDL and VAX RMS are provided, as are the procedures needed to perform tasks usually required in file-based application development.

Intended Audience

This document is intended for applications programmers and designers creating or maintaining applications programs that use VAX RMS files. It may also be read to gain a general understanding of the comprehensive set of file and record processing options available on a VAX/VMS or MicroVMS system.

Structure of This Document

This guide contains ten chapters and one appendix.

- Chapter 1 provides general information on the use of files on a VAX/VMS system, including an overview of available media, VAX RMS, FDL, and resource requirements.
- Chapter 2 describes the file organizations and record access modes to help you choose the correct file organization for your application. For Version 4.0, this material was located in the first part of the chapter formerly titled "Designing Files."
- Chapter 3 discusses general performance considerations and specific trade-offs you can make in the design of your application. For Version 4.0, this material was located in the second part of the chapter formerly titled "Designing Files."
- Chapter 4 describes procedures necessary to create files, populate files with records, and protect files.
- Chapter 5 describes file specifications and the procedures needed to use them.

- Chapter 6 describes the rules of file specification parsing and advanced file specification use. Information about rooted directories is also provided.
- Chapter 7 describes file sharing and buffering, including record locking and the use of global buffers.
- Chapter 8 describes aspects of record processing, including record access modes; synchronous and asynchronous record operations; and retrieving, inserting, updating, and deleting records.
- Chapter 9 describes how to specify run-time options and summarizes the run-time options available when a file is opened and closed and when records are retrieved, inserted, updated, and deleted.
- Chapter 10 describes procedures needed to maintain properly tuned files, with the emphasis on efficiently maintaining indexed files.
- Appendix A describes the algorithms used by the Edit/FDL Utility.

A glossary of terms is also provided.

Associated Documents

The reader should be familiar with the information in the following documents:

- The *Introduction to VAX/VMS* describes the use of VAX/VMS for a general audience.
- Programmers should be familiar with the appropriate documentation for the VAX language the application will be written in.

Related reference information is available in the following documents:

- The *VAX/VMS File Definition Language Facility Reference Manual* contains information about the FDL facility.
- The *VAX/VMS Convert and Convert/Reclaim Utility Reference Manual* contains information about the Convert Utility that is often used in conjunction with file applications.
- The *VAX/VMS DCL Dictionary* contains information (with examples) about using DCL commands that define system or process defaults, set file protection, or define logical names.

- The *VAX Record Management Services Reference Manual* contains information about calling VAX RMS services directly and about the control block options that are available. This book describes the VAX RMS control blocks that define arguments for VAX RMS service calls performed through language statements or directly from user programs. This document also includes special information for VAX MACRO users, including descriptions of VAX MACRO service and control block macros and examples of VAX MACRO programs.
- The *VAX/VMS Utility Routines Reference Manual* contains information about calling FDL routines and Convert routines. It also includes appropriate programming examples.
- Programmers using DECnet to access remote files may need to consult the *VAX/VMS Networking Manual* to determine what types of operations are supported for remote files on non-VAX/VMS systems. Information on accessing remote files on a VAX/VMS system can be obtained from the description of the appropriate FDL attributes in the *VAX/VMS File Definition Language Facility Reference Manual* and the description of the appropriate control block fields in the *VAX Record Management Services Reference Manual*.

Conventions Used in This Document

Convention	Meaning
<code>RET</code>	A symbol with a one- to six-character abbreviation indicates that you press a key on the terminal, for example, <code>RET</code> .
<code>CTRL/x</code>	The phrase CTRL/x indicates that you must press the key labeled CTRL while you simultaneously press another key, for example, CTRL/C, CTRL/Y, CTRL/O.
\$ SHOW TIME 05-JUN-1985 11:55:22	Command examples show all output lines or prompting characters that the system prints or displays in black letters. All user-entered commands are shown in red letters.
\$ TYPE MYFILE.DAT . . .	Vertical series of periods, or ellipsis, mean either that not all the data that the system would display in response to the particular command is shown or that not all the data a user would enter is shown.

Preface

Convention	Meaning
file-spec,...	Horizontal ellipsis indicates that additional parameters, values, or information can be entered.
[logical-name]	Square brackets indicate that the enclosed item is optional. (Square brackets are not, however, optional in the syntax of a directory name in a file specification or in the syntax of a substring specification in an assignment statement.)
quotation marks apostrophes	The term quotation marks is used to refer to double quotation marks ("). The term apostrophe (') is used to refer to a single quotation mark.

New and Changed Features

The following new features apply to the information discussed in this guide:

- You can now access indexed records in both ascending and descending sort order. In some instances, you may wish to define two keys for the same record field, one ascending type and one descending type. This permits you to process records sequentially in either *forward* or *reverse* order, except for duplicate key values.
- Under VAX/VMS Version 4.4, VAX RMS supports full file sharing, record locking and the use of global buffers for all sequential files. Prior to VAX/VMS Version 4.4, this capability was restricted to sequential files with fixed-length, 512-byte records.

In addition, the *connect at end of file* option has been enhanced for shared sequential files only. With VMS Version 4.4, the user program is able to maintain the next record position at the current end of file for sequential \$PUT operations, irrespective of the positioning done by other processes sharing the file.

This capability is typically used to log events to a shared central file.

- A logical name that is defined with the concealed attribute no longer must translate directly to a physical device but can translate to another logical name as shown in the following example:

```
$ DEFINE MY_ROOT DISK$WORK:[ROOT.]/TRANSLATION=CONCEALED
```


1

Introduction

As technology has progressed, so has the volume of information that must be maintained. For instance, business and industrial concerns have compiled many types of data files about a wide range of subjects. For a long time, all these data files were stored efficiently on paper in desk drawers and filing cabinets. But, as these paper files grew, it often took longer to locate the needed data than to create it in the first place. Storage space also became a problem.

As the need for saving data increased, a better medium was required to store and retrieve data quickly, reliably, and economically. The computer (and computerized filing systems) filled this need by using the following storage media:

- Punched cards
- Paper tape
- Magnetic tape
- Disk

At first, computerized files consisted of collections of punched cards. Cards provide a convenient means of grouping related pieces of information. This information might represent, for example, a business event, such as a purchase or sale of office furniture. Or, in an engineering environment, the information could represent the source data used in equations. This information, grouped on a single card, represents a record of that event. Records of similar events, grouped together, constitute a file.

As a storage medium, cards are relatively slow to process because they allow only sequential access. Sequential access means that the search for a record starts at the beginning of the file and proceeds in order through each record. At times, when the needed record (or group of records) is near the end of the file, the search wastes computer processing time, especially if the file is large.

The introduction of magnetic tape provided both a nonmechanical storage medium and a means for storing data (input) and for retrieving data (output), commonly referred to as input/output (I/O). Compared with punched cards, magnetic tape offers virtually unlimited offline data storage (data that is not immediately available to a user) because a single tape reel can accommodate a large amount of data and you can use as many reels as necessary.

Compared to disk storage, magnetic tape is generally less expensive for use in offline storage and for transporting data between computer systems. Thus, magnetic tape is often used when data must be transported between computers made by different manufacturers or when data cannot be sent over public networks because of security reasons.

A primary use of magnetic tape is for offline storage of disk archives (backup) which is done periodically to prevent the loss of data. Because fast access is not usually a requirement for backup data, it is well suited to magnetic tape storage.

For online processing however, magnetic tape has the same drawback as punched cards; that is, a sequential file organization that limits the capability for fast access.

In contrast to magnetic tape storage, disk storage allows faster data access while providing the same virtually limitless storage capacity. Disks provide faster access because the computer can locate files and records selectively without first searching through intervening data. This faster access time, therefore, makes disks more appropriate for online file processing applications.

This chapter illustrates basic data management concepts and how they are applied by the VAX Record Management Services (VAX RMS). VAX RMS is the data management subsystem of the VAX/VMS operating system. In combination with the VAX/VMS operating system, VAX RMS allows efficient and flexible storage, retrieval, and modification of data on disks, magnetic tapes, and other devices. VAX RMS may be implemented directly at the VAX MACRO level, or indirectly by way of the File Definition Language (FDL) utility. Higher-level languages may also implement VAX RMS through program-specific processing options. Although VAX RMS supports devices such as line printers, terminals, and card readers, the purpose of this guide is to introduce you to VAX RMS record keeping on magnetic tape and disk.

Section 1.1 presents file processing concepts. Section 1.2 discusses physical disk and tape concepts. Section 1.3 covers volume, directory, and file protection. Section 1.4 presents an overview of the File Definition Language, VAX RMS data structures, and the VAX RMS Services. Section 1.5 discusses the VAX RMS utilities. Section 1.6 discusses process and system requirements associated with a file application environment.

1.1 File Concepts

The file concepts discussed here include:

- File organization
- Records structures
- Fields
- Bytes and bits
- Access modes
- Record formats

A file is an organized collection of data stored on a mass storage volume, such as a disk, and processed by a central processing unit (CPU). Data files are organized to accommodate the processing of data within the file by an application program. The basic unit of electronic data processing is the *record*. A record is a collection of related data that your program treats as a whole. For example, all the information on an employee, such as name, street address, city, and state, constitutes a personnel record. Records are made up of *fields* or *items*, which are sets of contiguous bytes. For example, a first name or a city might be a field. A *byte* is a group of binary digits (bits), which are used to represent a single character. You can also think of a field or an item as a group of bytes in a record that are related in some way.

The records in a file must be formatted uniformly. That is, they must conform to some defined arrangement of the fields that make up each record including the lengths of the fields, the location of each field in the record, and the type of data (character strings or binary integers, for instance) in each field. In order to process the data in a file, an application must know the arrangement of the record fields, especially if the application intends to modify existing records or to add new records to the file.

The *file organization* is the arrangement of data within a file. The file organization together with the applicable storage medium determine what techniques are used to store and retrieve data. Currently, VAX RMS supports three file organizations, *sequential*, *relative*, and *indexed*.

Introduction

Sequential	Records are arranged one after the other.
Relative	Each record occupies a cell of equal length within a bucket. Each cell is assigned a successive number, called a <i>relative record number</i> , which represents the cell's position relative to the beginning of the file.
Indexed	Records can either be retrieved randomly or sequentially in sorted order.

The manner in which your program stores and retrieves the data it processes is called the *record access mode*. VAX RMS supports two access modes.

Sequential Access	Records are stored or retrieved one after another starting at a particular point in the file and continuing in order through the file.
Random Access	Records are stored and retrieved by key, by relative record number, or by file address. Random access by key or by relative record number depends upon the file organization. Indexed file records are stored and retrieved by a <i>key</i> in the data record; relative file records are retrieved by their <i>relative record numbers</i> . When a record is accessed randomly by its file address, the distinction is made by its unique location in the file; that is, its <i>record file address</i> (RFA).

The *record format* indicates the way all records in a file appear physically on the recording surface of the storage medium and is defined in terms of record length. VAX RMS supports four record formats:

Fixed length	All file records are the same length.
Variable length	Records vary in length.
Variable with fixed-length control	Records do not have to be the same length, but each includes a fixed-length control field that precedes the variable-length data portion.
Stream	Records are delimited by special characters or character sequences called <i>terminators</i> . Records with stream format are interpreted as a continuous sequence, or stream, of bytes. The carriage return and the line feed characters are commonly used as terminators.

When you create a VAX RMS file, you specify the file storage medium and the file and record characteristics directly through your application program or indirectly using an appropriate utility. Chapter 2 outlines VAX RMS file organizations, record access modes, and record characteristics in detail.

After VAX RMS creates the file in accordance with the specified file and record characteristics, your application program must consider these characteristics when storing, retrieving and modifying file records. Chapters 4 and 8 outline techniques for creating, storing, retrieving, and modifying data records in a file.

1.1.1 Disk Structures

This section describes disk structures as an aid to understanding how a disk may be configured to enhance data access for improved performance. Disk structures may be defined as either logical or physical and the two types of structure interact upon each other to some degree. That is, you cannot manipulate a logical structure without considering the effect on a corresponding physical structure.

1.1.1.1 Logical Structures

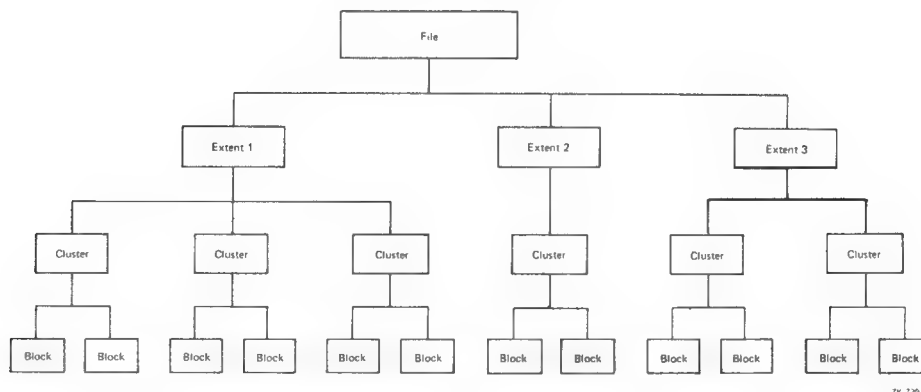
VAX RMS disk files reside on *Files-11 On-Disk Structure* disks. Files-11 On-Disk Structure defines the organization of the disk external to the files. By contrast, VAX RMS defines the internal organization of the files and the methods of accessing file data.

Two distinct types of disk structuring are used with VAX RMS, Files-11 On-Disk Structure Level 1 and Files-11 On-Disk Structure Level 2. Files-11 On-Disk Structure Level 1 is the default for the RSX-11M, RSX-11D, and RSX-11-PLUS operating systems; VAX/VMS utilizes Files-11 On-Disk Level Structure 2. The primary difference between the two structures is that Files-11 On-Disk Structure Level 2 incorporates control capabilities that permit added features including volume sets (described later).

The term Files-11 On-Disk Structure refers to logical ordering of the disk using the following disk structures listed in order of ascending hierarchy:

- Blocks
- Clusters
- Extents
- Files
- Volumes
- Volume Sets

Figure 1-1 shows the hierarchy of blocks, clusters, extents, and files in the Files-11 On-Disk Structure.

Figure 1-1 Files-11 On-Disk Structure Hierarchy

The next higher level of Files-11 On-Disk Structure is the *volume* (not illustrated) which is defined as the ordered set of blocks that comprise a disk. However, a volume may in fact include several disks that together make up a structure called a *volume set*. Because a volume set consists of two or more related volumes, the system treats it as a single volume.

Note

For this description, the terms disk and volume are used interchangeably.

The smallest addressable logical structure on a Files-11 On-Disk Structure disk is a *block*. A block consists of 512, 8-bit bytes and one or more blocks may be treated as a single unit for transfer between a Files-11 On-Disk Structure disk and memory.

VAX RMS allocates disk space for new files or extended files using multiblock units called a *clusters*. The number of blocks in a disk cluster is established by the system manager or operator when a volume is initially prepared for use (initialized).

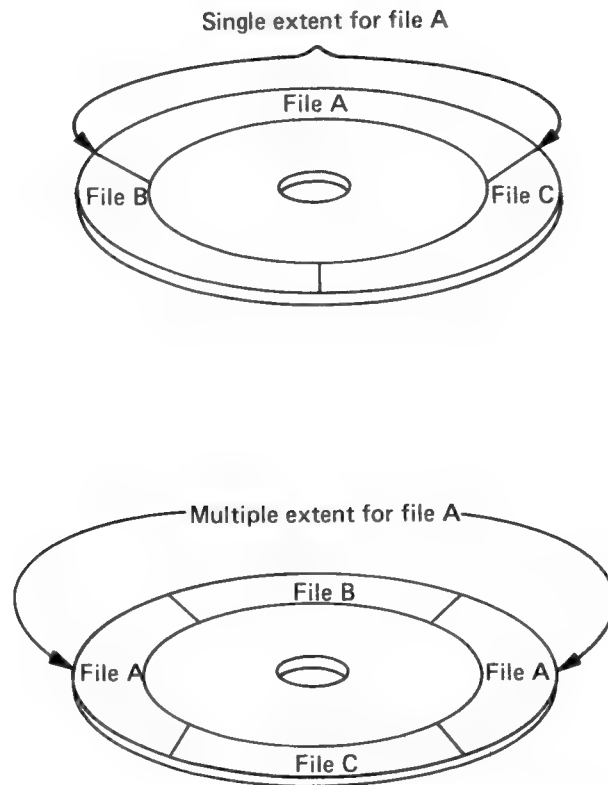
Note

Do not confuse a disk cluster with a VAXcluster, a configuration of VAX/VMS systems.

Clusters may or may not be contiguous on a disk and sizes may range from 1 to 65,535 blocks. Generally, a system manager assigns a disk with a relatively small number of blocks a small cluster size, and relatively larger disks a larger cluster size to minimize the overhead for disk space allocation.

Contiguous clusters allocated to a particular file are called an *extent* and these may contain all or part of a file. If enough contiguous disk space is available, the entire file is allocated as a single extent. Conversely, if there is not enough contiguous disk space, the file is allocated and extended using several extents that may be physically scattered on the disk as shown in Figure 1-2.

Figure 1-2 File Extents



ZK-738-82

With VAX RMS, you can exercise varying degrees of control over file space allocation. At one extreme, you can specify the number of blocks to be allocated and their precise location on the volume. At the other extreme, you can allow VAX RMS to handle all disk space allocation details automatically. As a compromise, you might specify the size of the initial space allocation and have VAX RMS determine the amount of space allocated each time the file is extended. You can also specify that unused space at the end of the file is to be deallocated from the file, making that space available to other files on the volume.

When you need a large amount of file storage space, you can combine several Files-11 On-Disk Structure volumes into a *volume set* with file extents located on different volumes in the set. You need not specify a particular volume in the set to locate or create a file, but you may improve performance if you explicitly specify a volume for a particular allocation request.

1.1.1.2 Physical Structures

For performance reasons, you should be aware of certain physical aspects of a disk.

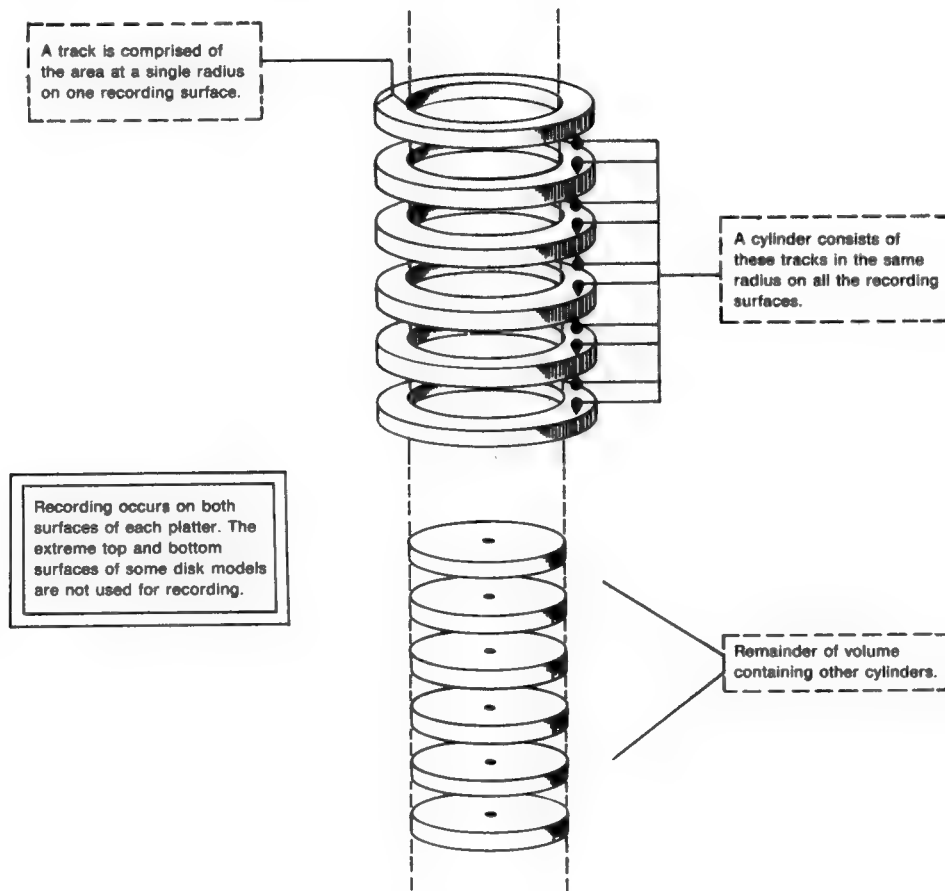
A disk (or volume) consists of one or more *platters* that spin at very high, constant speeds and usually contain data on both surfaces (upper and lower). A *disk pack* is made up of two or more platters having a common center.

Data is located at different distances from the center of the platter and is stored or retrieved using read/write heads that move to access data at various radii from the platter's center. The time required to position the read/write heads over the selected radius (referred to as a *track*) is called *seek time*. Each track is divided into 512-byte structures called *sectors*. The time required to bring the selected sector (logical block) under the read/write heads at the selected radius (track) is called the *rotational latency*. Because seek time usually exceeds the rotational latency by a factor of 2 to 4, related blocks (sectors) should be located at or near the same track to obtain the best performance when transferring data between the disk and VAX RMS-maintained buffers in memory. Typically, related blocks of data might include the contents of a file or several files that are accessed together by a performance-critical application.

Another physical disk structure is called a *cylinder*. A cylinder consists of all tracks at the same radius on all recording surfaces of a disk.

Figure 1-3 illustrates the relationship between tracks and cylinders.

Figure 1-3 Tracks and Cylinders



ZK-740-82

Because all of the of the blocks in a cylinder can be accessed without moving the disk's read/write heads, it is generally advantageous to keep related blocks in the same cylinder. For this reason, when choosing a cluster size for a large-capacity disk, a system manager should consider one that divides evenly into the cylinder size.

1.1.1.3 Files-11 On-Disk Structure Index File

Each Files-11 On-Disk Structure volume has an *index file* containing control information that Files-11 On-Disk Structure software maintains for VAX RMS. This information is transparent to your program but VAX RMS uses it to give your program access to individual file records. The *Guide to VAX/VMS Disk and Magnetic Tape Operations* contains details on the various Files-11 On-Disk Structure control structures; but the discussion here is limited to two components of the index file, the *home block* and the *file headers*.

The home block is usually the second block on a volume and it identifies the disk as a Files-11 On-Disk Structure volume. If for some reason the home block cannot be read (physically unusable), an alternative block will be selected for use as the home block. This block provides specific information about the volume and default values for files on the volume. The following items are in the home block:

- The volume name
- Information to locate the remainder of the index file
- The maximum number of files that can be present on the volume at any one time
- The user identification code (UIC) of the owner of the volume
- Volume protection information (specifies which users can read and/or write the entire volume)

Files-11 On-Disk Structure volumes contain several copies of the home block to ensure that the accidental destruction of this information does not affect VAX RMS ability to locate other files on the volume.

The bulk of the volume's index file consists of file headers, each of which describes one file on the volume. File headers include the following information:

- File ownership
- Protection
- Creation date
- Creation time

Each file header contains a list of the extents that make up the file, describing where the file is physically located on the volume. If a file has a large number of extents, it may be necessary to have multiple file headers for locating them. In this case, a file identifier number is associated with each file header.

When you create a file, you normally specify a name which VAX RMS assigns to the file on a Files-11 On-Disk Structure volume. VAX RMS places the file name and file identifier associated with the newly-created file in a directory that contains an entry defining the location for each file. When you access the file, you supply the file name, which in turn supplies a path to the file identifier through the directory entry. The file identifier points to the location of the file header, which contains a listing of the extent or extents that locate the data.

1.1.2 Tape Concepts

You can also use magnetic tape as a file storage medium with VAX RMS. VAX RMS supports the standard magnetic tape structure defined by American National Standard X3.27-1977, *Magnetic Tape Labels and Record Formats for Information Interchange*.

Data records are organized on magnetic tape in the order in which they are entered; that is, sequentially.

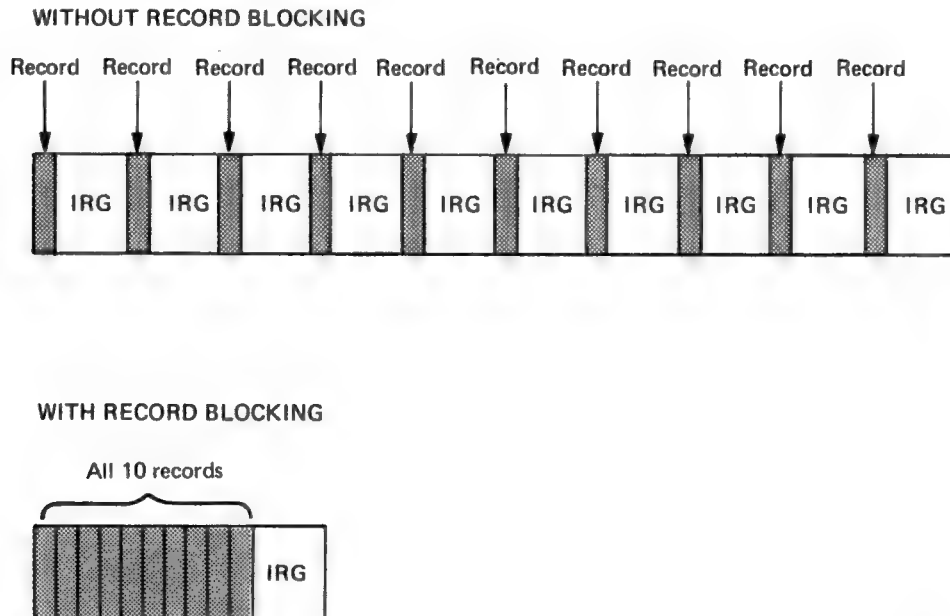
Characters of data on magnetic tape are measured in *bits per inch* (bpi). This measurement is called *density*. A 1600-bpi tape can accommodate 1600 characters of data in 1 inch of recording space. A tape has 9 parallel tracks containing 8 bits and 1 parity bit. A parity bit is used to check for data integrity using a scheme wherein each character contains an odd number of marked bits, regardless of its data bit configuration.

Even though a tape may have a density of 1600 bpi, there are not always 1600 characters on every inch of magnetic tape because of the *interrecord gap* (IRG). The IRG is an interval of blank space between data records that is created automatically when records are written to the tape. The IRG is a breakpoint on the tape, which allows the tape unit to decelerate, and stop if necessary, after a record operation and then accelerate to working speed before the next record operation.

Each IRG is approximately 0.6 inch in length. Writing an 80-character record at 1600 bpi requires 0.05 inch of space. The IRG, therefore, requires twelve times more space than the data with a resultant waste of storage space. VAX RMS can reduce the size of this wasted space by using a record blocking technique that groups individual records into a block and places the IRG after the block rather than after each record. (A block on disk is different from a block on tape. On disk, a block is fixed at 512 bytes; on tape, you determine the size of a block.) However, record blocking requires more buffer space to be allocated for your program resulting in an increased need for memory. The greater the number of records in a block, the greater the buffer size requirements. You must determine the point at which the benefits of record blocking cease based on the configuration of your computer system.

The example in Figure 1-4 shows how space can be saved by record blocking. Assume that a 1600-bpi tape contains 10 records that are not grouped into blocks. Each record is 160 characters long (0.1 inch at 1600 bpi) with a 0.6-inch IRG after each record; this uses 7 inches of tape. However, placing the same 10 records into 1 block uses only 1.6 inches of tape (1 inch for the data records and 0.6 inch for the IRG). example of record blocking.

Figure 1-4 Interrecord Gaps



ZK-741-82

Record blocking also increases the efficiency of the flow of data into the computer. For example, 10 unblocked records require 10 separate physical transfers, while 10 records placed into a single block require only 1 physical transfer. Moreover, a shorter length of tape is traversed for the same amount of data to reduce operating time.

Like disk data, magnetic tape data is organized into files. When you create a file on tape, VAX RMS writes a set of header labels on the tape immediately preceding the data blocks. These labels contain information such as the user-supplied file name, creation date, and expiration date. Additional labels,

called trailer labels, are also written following the file. To access a file on tape by the file name, the file system searches the tape for the header label set that contains the specified file name.

When the data blocks of a file or related files do not physically fit on one volume (a reel of tape), the file is continued on another volume, creating a multivolume tape file that contains a volume set. When a program accesses a volume set, all volumes in the set are searched. For additional information about magnetic tapes, see the *Guide to VAX/VMS Disk and Magnetic Tape Operations*.

1.2 Volume Protection

The system protects data on disk and tape volumes to make sure that no one accesses the data accidentally or without authorization. For disk volumes, the system provides protection at the file, directory, and volume levels. For tape volumes, the system provides protection at the volume level only.

In addition to protecting the data on mounted volumes, the system provides device protection coded into the home block of the disks and tapes. See Section 1.2.1 for more information.

In general, you can protect your disk and tape volumes with user identification codes (UICs) and access control lists (ACLs). The standard protection mechanism is UIC-based protection. The use of optional access control lists permits you to customize security for a file or a directory.

UIC-based protection is determined by an owner UIC and a protection code, whereas ACL-based protection is determined by a list of entries that grant or deny access to specified files and directories.

Note

You cannot use ACLs with magnetic tape files or with files accessed over a network.

When you try to access a file having an ACL, the system uses the ACL to determine whether or not you have access to the file. If the file has an ACL but the ACL does not explicitly allow or refuse you access, the system uses the UIC-based protection to determine access. If the file has no associated ACL, the system uses UIC-based protection to determine access. (See the *Guide to VAX/VMS System Security* for additional information on system security.)

For detailed information on protecting your files, directories, or volumes, see Section 3.3.

1.3 Record Management Services

VAX Record Management Services (VAX RMS) is the file and record access subsystem of the VAX/VMS operating system. In combination with the VAX/VMS operating system, VAX RMS allows efficient and flexible data storage, retrieval, and modification for disks, magnetic tapes, and other devices.

You can use VAX RMS from low-level and high-level languages. If you use a high-level language, it may not be easy or feasible to use the VAX RMS services directly, because you must allocate control blocks and access fields within them. Instead, you can rely on certain processing options of your language's I/O statements or upon a specialized language provided as an alternative to using VAX RMS control blocks directly, the *File Definition Language*.

If you use a low-level language, however, you can either use VAX RMS services directly or you may use the File Definition Language.

1.3.1 File Definition Language

The File Definition Language, or simply FDL, is a special-purpose language that you can use to specify file creation and run-time access characteristics of the file. FDL is particularly useful when you are using a high-level language or when you want to ensure that you create properly tuned files. Properly tuned files can be created from either an existing file or a new design for a file. The performance benefits of using properly tuned files can greatly affect application and system performance, especially when large indexed files are used.

FDL allows you to use all of the creation-time and many of the run-time capabilities of the VAX RMS control blocks such as the *file access block* (FAB), the *record access block* (RAB), and the *extended attribute blocks* (XABs).

For more information on FDL, see Section 3.1.2.

1.3.2 VAX RMS Data Structures

VAX RMS data structures, or control blocks, generally fall into two groups: those whose information pertains to files and those whose information pertains to records.

To exchange file-related information with VAX RMS file services, you use a control block called a file access block (FAB). You use the FAB to define file characteristics, file specifications, and various run-time options. There are a number of fields in the FAB, each of which is identified by a symbolic offset value. For instance, the allocation quantity for a file is specified in a longword-length field having a symbolic offset value of FAB\$L_ALQ in the FAB. The symbol FAB\$L_ALQ indicates the number of bytes from the beginning of the FAB where the field begins.

To exchange record-related information with VAX RMS record services, you use a control block called a record access block (RAB). You use the RAB to define the location, type, and size of the input and output buffers, the record access mode, certain tuning options, and other information. The symbolic offset values for the RAB fields have the prefix RAB\$ to differentiate them from the values used to identify FAB fields. These names have the same general format, where the letter following the dollar sign indicates the field length and the letters following the underscore are a mnemonic of the field's name. For example, the multibuffer count field specifies the number of VAX RMS buffers to be used for I/O and has the symbolic offset value RAB\$B_MBF, which indicates the number of bytes from the beginning of the RAB to the start of the field.

Where applicable, VAX RMS uses control blocks called *extended attribute blocks*, or XABs, together with FABs and RABs to support the exchange of information with VAX RMS services. For example, a Key Definition XAB specifies the characteristics for each key within an indexed file. The symbolic offset values for XAB fields have the common prefix XAB\$.

For more information on VAX RMS control blocks, see Chapter 4.

1.3.3 VAX RMS Services

Because VAX RMS performs operations related to files and records, services generally fall into one of two groups:

- Services that support file processing. These services create and access new files, access (or open) previously created files, extend the disk space allocated to files, close files, get file characteristics, and perform other functions related to the file.

- Services that support record processing. These services get (extract), find (locate), put (insert), update (modify), delete (remove) records and perform other record operations.

For more information on the VAX RMS services, see Chapters 7 and 8.

1.4 VAX RMS Utilities

The following are VAX RMS file-related utilities:

- The Analyze/RMS_File Utility
- The Convert Utility
- The Convert/Reclaim Utility
- The Create/FDL Utility
- The Edit/FDL Utility

These utilities let you design, create, populate, maintain, and analyze data files that can use the full set of VAX RMS create-time and run-time options. They help you create efficient files that use a minimum amount of system resources, while decreasing I/O time.

See the appropriate Utility Manual for more information.

1.4.1 The Analyze/RMS_File Utility

The Analyze/RMS_File Utility (ANALYZE/RMS_FILE) can perform five functions:

- Inspect and analyze the internal structure of a VAX RMS file
- Generate a statistical report on the file's structure and use
- Allow you to explore the file's internal structure interactively
- Generate an FDL file from a VAX RMS file
- Generate a summary report on the file's structure and use

ANALYZE/RMS_FILE is particularly useful in generating an FDL file from an existing data file that you can then use with the Edit/FDL Utility (also called the FDL editor) to optimize your data files. Alternatively, you can provide general tuning for the file without the FDL editor.

To invoke the Analyze/RMS_File Utility, use the DCL command

```
ANALYZE/RMS_FILE file-spec[,...]
```


The *file-spec* parameter lets you select the data file you want to analyze. For more information on this utility, see Chapter 10 and the *VAX/VMS Analyze/RMS File Utility Reference Manual*.

1.4.2 The Convert Utility

The Convert Utility (CONVERT) copies records from one or more files to an output file, optionally changing the record format and file organization to that of the output file.

CONVERT is particularly useful in the tuning cycle of a file. After you have analyzed and optimized the file, you can use CONVERT to create a new file having the new, optimized characteristics and to copy the records in the old file to the newly created file. You can also use CONVERT to reformat an indexed file that has had many record insertions or deletions.

To invoke the Convert Utility, use the DCL command

```
CONVERT input-file-spec[...] output-file-spec
```

The *input-file-spec* parameter lets you specify the file or files you want to convert and the *output-file-spec* parameter lets you specify a destination file for the converted records.

Figure 1-5 shows how CONVERT creates data files and loads them with converted records from an input file.

For more information on this utility, see Chapter 4 and the *VAX/VMS Convert and Convert/Reclaim Utility Reference Manual*.

1.4.3 The Convert/Reclaim Utility

The Convert/Reclaim Utility reclaims empty buckets in Prolog 3 indexed files so that new records can be added to them. By reclaiming these buckets, you may be able to avoid extending a file and conserve disk space for other files.

To invoke the Convert/Reclaim Utility, use the DCL command

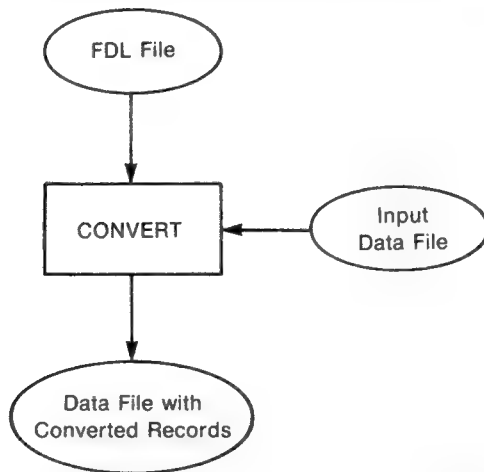
```
CONVERT/RECLAIM file-spec
```

The *file-spec* parameter specifies the Prolog 3 indexed file in which you want to reclaim empty buckets.

The Convert/Reclaim Utility does an "in-place" reorganization of the file in contrast to the Convert Utility which creates a new file from the old file. For this reason, the Convert/Reclaim Utility is more appropriate to being used for large disk files where space is limited. Prior to using the Convert/Reclaim Utility, be sure to back up the file.

For more information on this utility, see Chapter 4 and the *VAX/VMS Convert and Convert/Reclaim Utility Reference Manual*.

Figure 1-5 CONVERT Creates Data Files



ZK-946-82

1.4.4 The Create/FDL Utility

The Create/FDL Utility (CREATE/FDL) uses the specifications in an existing FDL file to create a new, empty data file.

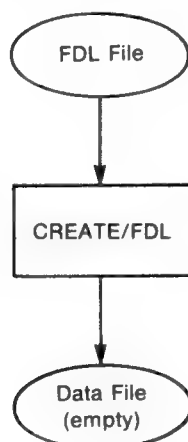
To invoke this utility, use the DCL command

CREATE/FDL=fdl-file-spec [file-spec]

The *fdl-file-spec* parameter specifies the FDL file from which to create the data file. The *file-spec* parameter gives you the option of assigning a file specification to the created file.

Figure 1-6 shows how CREATE/FDL creates empty data files from the specifications in an FDL file. For more information on this utility, see Chapter 4 and the *VAX/VMS File Definition Language Facility Reference Manual*.

Figure 1-6 CREATE/FDL Creates Empty Data Files



ZK-945-82

1.4.5 The Edit/FDL Utility

The Edit/FDL Utility (EDIT/FDL) creates and modifies files that contain specifications for VAX RMS data files. The specifications are written in the File Definition Language (FDL), and the files are called FDL files.

A completed FDL file is an ordered sequence of file attribute keywords and their associated values. By using an FDL file to specify the characteristics of a data file, you can use most of the VAX RMS capabilities without having to access the VAX RMS control blocks directly.

EDIT/FDL recognizes proper FDL syntax and informs you immediately of syntax errors. It permits you to model a data file, altering the values of attributes to see the effects of different attribute values on the design of the file. Using EDIT/FDL, you can experiment with attributes that are critical to the record-processing performance of the file and you can calculate optimum file size.

For example, the depth of an index is an important consideration in designing an indexed file, and bucket size is one variable that determines the number of levels. EDIT/FDL can show the effects of varying the bucket size on the index depth to help you choose the optimum bucket size.

To invoke this utility, use the DCL command

`EDIT/FDL fdl-file-spec`

The *fdl-file-spec* parameter specifies the FDL file that you want to create, modify, or optimize.

For more information, see the *VAX/VMS File Definition Language Facility Reference Manual*.

1.5 Process and System Resources for File Applications

To use VAX RMS files efficiently, your application requires various process and system resources. You may have to adjust specific resources and quotas for the process running a file application. You should coordinate process and system requirements with your system manager during the application design (or redesign) procedure before implementing the VAX RMS options that require the additional resources for the application. In some cases, the system manager may want to order additional memory or disk drives to ensure that sufficient system resources are available.

1.5.1 Memory Requirements

One of the most important ways to improve application performance is to allocate larger buffer areas or more buffers for an application. As described in Chapter 7, the number of buffers and the size of buckets and blocks can be fine tuned on the basis of the way the file will be accessed. For indexed files, the index structure and other factors must also be considered.

When a file is opened or created, VAX RMS maintains the specified buffers and control structures that are charged against process memory use. Memory use generally increases with the number of files to be processed at the same time. The amount of memory needed for I/O buffers can vary greatly for each file but the amount of memory needed for control structures is fairly constant.

The memory use (working set) of a process is governed by three SYSGEN parameters:

- Working set default (WSDEFAULT) specifies the initial size of the working set in 512-byte pages.
- Working set quota (WSQUOTA) specifies the maximum size, in pages, that the working set can grow to unless physical memory pages are available and a larger working set extent value is specified.
- Working set extent (WSEXTENT) specifies the maximum number of pages in a working set including free pages of memory.

These values can ensure that the process has sufficient memory to perform the application with minimum paging. For a complete description of these limits, see the *VAX/VMS System Manager's Reference Manual*.

In addition to process requirements, you may want a shared file to use global buffers to avoid needless I/O when the desired block is already in memory. The use of global buffers is governed by the following SYSGEN parameters:

- VAX RMS global buffer quota (RMS_GBLBUFQUO) limits the maximum number of VAX RMS global buffers in use on a system simultaneously regardless of the number of users or files.
- Global sections (GBLSECTIONS) specifies the maximum number of global sections in use simultaneously on the system.
- Global pages (GBLPAGES) limits the number of global page table entries in use simultaneously on the system.
- Global page-file sections (GBLPAGFIL) limits the number of system-wide pages for global page-file sections or scratch global sections in use simultaneously on the system.

When DCL opens a process-permanent file, VAX RMS places internal structures for this file in a special portion of P1 space called the *process I/O segment*. The size of this segment is determined by the system and cannot be expanded dynamically. If DCL tries to open a file and there is not enough space in the process I/O segment for the internal structures, you will receive an error and the file will not be opened.

For a complete description of these parameters, see the *VAX/VMS System Generation Utility Reference Manual*.

1.5.2 Process Asynchronous I/O Limits

If you anticipate asynchronous record I/O, you should examine the following limits:

- Asynchronous system trap limit (ASTLM)
- Buffered I/O limit (BIOLM)
- Direct I/O limit (DIOLM)

For a complete description of these limits, see the *VAX/VMS System Manager's Reference Manual*.

1.5.3 Process Record-Locking Quota

If the application is to access a shared file for which record modifications or additions are allowed, the process enqueue quota should be examined. The need to increase the process enqueue quota (ENQLM) varies with the number of records that may be simultaneously locked, multiplied by the number of open files.

For a complete description of this quota, see the *VAX/VMS System Manager's Reference Manual*.

1.5.4 Process Open File Limit

The number of files that a process may have open simultaneously is governed by the open file limit (FILLM).

For a complete description of this limit, see the *VAX/VMS System Manager's Reference Manual*.

1.6 Setting Process and System Defaults

Your application can use certain defaults if you do not explicitly specify a value when you create or open a file. After implementing values you specify directly in the FAB or RAB, VAX RMS applies file defaults you have set such as the file default extension quantity. If more values are needed, VAX RMS applies process defaults and then volume or system defaults in that order. Typically, a system default is a system parameter that you set with the System Generation Utility (SYSGEN). (For more information, see the *VAX/VMS System Generation Utility Reference Manual*.)

You can use the DCL command SET RMS_DEFAULT to set process or system defaults for such items as buffering and you can use the SET FILE command for file defaults. Refer to the *VAX/VMS DCL Dictionary* for additional information on these commands.

2

Choosing A File Organization

When you write an application program, you want the program to input data, process it, store it, update it if necessary, and output it at the right time in the right format. Moreover, the program should perform these functions quickly and accurately.

To achieve this objective, your design should consider the structure of your data files and the data processing capabilities available to you through VAX RMS.

You should consider these factors when you write the application, especially if you have large files with many users simultaneously accessing the files, or a high level of file activity where many records are stored, retrieved, updated, or deleted in a given time period.

You may subsequently reconsider these factors if, after using the application, you are not satisfied with its performance.

This chapter describes the various aspects of file design and structure to help you make the first important design decision: selecting a file organization. Section 2.1 covers record access modes and formats. Section 2.2 describes file concepts and organization.

See Chapter 3 for a description of performance criteria that will help you to evaluate the performance of your data files.

All of the VAX RMS capabilities described in this chapter are available at the VAX MACRO programming level and many are available to higher-level programming languages that use FDL as an intermediary to the VAX RMS control blocks. (See the descriptions of the FDL routines in the *VAX/VMS Utility Routines Reference Manual* for details.) If you intend to use VAX RMS from a higher-level language, refer to your language manual to determine the VAX RMS capabilities available to you.

2.1 Record Concepts

In considering the structure of your data files, you should note that a file is an ordered collection of logically related records treated as a unit and one consideration should be the way records are transferred to your program from storage.

For disk files, the smallest unit of transfer is a *block* but records are usually transferred in multiple blocks using transfer units that are in great part dictated by file organization. If you use the sequential file organization, the multiblock run-time option allows multiple blocks to be transferred during a single I/O operation. For relative and indexed file structures, records are transferred using buckets. A *bucket* is a storage structure, consisting of 1 to 63 blocks that is used for building and processing relative and indexed files.

Another consideration in file design is the manner in which records are to be accessed, referred to as the record access mode. The record access mode specifies the way your program stores and retrieves file records.

The final consideration in designing files is the manner in which records are formatted. The program that creates a file specifies its record format and any program that access the file must conform to the defined record format. The format defines the number and length of each field thereby establishing the length of each file record. For example, a program to create records in a payroll file might use a record format containing the following fields:

- Employee name
- Social security number
- Pay rate
- Deductions

The next two sections describe VAX RMS record access modes and record formats, respectively.

2.1.1 Record Access Modes

VAX RMS provides two record access modes, sequential access and random access. Random access can be further cataloged as one of the three following modes:

- Random access by key value
- Random access by relative record number
- Random access by record file address

Choosing A File Organization

Although you cannot change the file organization once it is established at file-creation time, you can change the record access mode each time the file is used. In fact, you can change the record access mode each time you access a record in the file. For example, a relative file can be processed in sequential record access mode one time and in one of the random access modes the next time. Table 2-1 lists the combinations of record access modes and file organizations supported by VAX RMS.

VAX RMS uses the concept of a *record stream* to establish an access pathway to file records and to maintain the access environment. Important elements of this environment are the current record position and the current record access mode, both of which are instrumental in determining which record is to be accessed next. Note that a separate RAB is required to support each record stream in a multistream environment.

Table 2-1 Record Access Modes and File Organizations

Access Mode	File Organization		
	Sequential	Relative	Indexed
Sequential	Yes	Yes	Yes
Random by relative record number	Yes ¹	Yes	No
Random by key value	No	No	Yes
Random by record's file address	Yes ²	Yes	Yes

¹Permitted with fixed-length record format on disk devices only

²Permitted on disk devices only

The following sections describe the record access modes and the capability for switching from one mode to another during program execution.

2.1.1.1 Sequential Access

In sequential access mode, storage or retrieval begins at a designated point in the file and continues sequentially through the file. Unless you specify the starting point explicitly, or have established a starting point through a previous operation, VAX RMS begins accessing records at the start of the file.

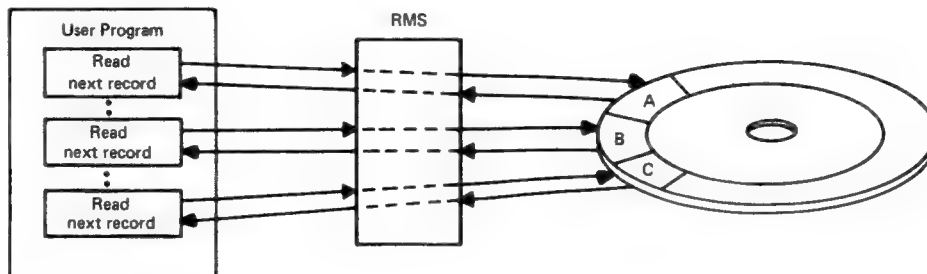
In the sequential access mode, your program issues a series of requests to VAX RMS to retrieve or store succeeding records in a file. Before acting on these requests, VAX RMS checks the file organization in order to determine how to proceed. The following sections describe how VAX RMS handles sequential access for each of the three file organizations.

Sequential Access to Sequential Files

In a sequential file, records are stored adjacent to each other. To retrieve a particular record within the file, your program must open the file and successively retrieve all records between the current record position and the selected record.

Figure 2-1 shows a disk volume surface. Each lettered section on the surface represents a record in a sequential file, beginning with record A. When the program requests sequential access to the file records, VAX RMS interprets each request in the context of the file's organization. Because this particular file is sequential, VAX RMS complies with each request (except for the first request) by accessing the record immediately following the previously accessed record. For example, after VAX RMS accesses record A, it updates the current-record position to record B in anticipation of the next request.

Figure 2-1 Sequential Access to a Sequential File



ZK-747-82

There are several limitations imposed by sequential access. If a program is accessing data sequentially, it cannot access a record that has passed unless

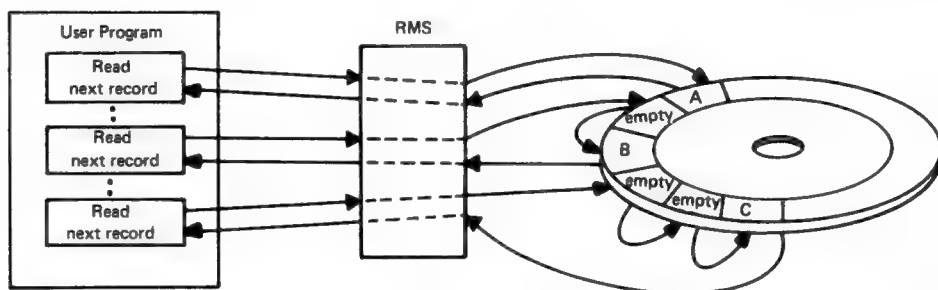
it reopens or rewinds the file, or unless it switches to a random access mode. (see Chapter 8 for details). A further limitation of sequential access is that you can add records to the end of the file only.

Sequential Access to Relative Files

Relative files may be accessed sequentially even if some of the fixed-length file cells are empty (because records were never stored in them or because records were deleted from them). If a cell is empty, VAX RMS ignores it and sequentially searches for the next cell that contains a record. For example, assume a relative file contains records only in cells 1, 3, and 6. VAX RMS responds to a sequential retrieval request by retrieving the record in cell 1, then the record in cell 3, and finally the record in cell 6.

Figure 2-2 shows how VAX RMS checks each cell, ignores an empty cell when it finds one, and then checks the next cell for a record.

Figure 2-2 Sequentially Retrieving Records in a Relative File



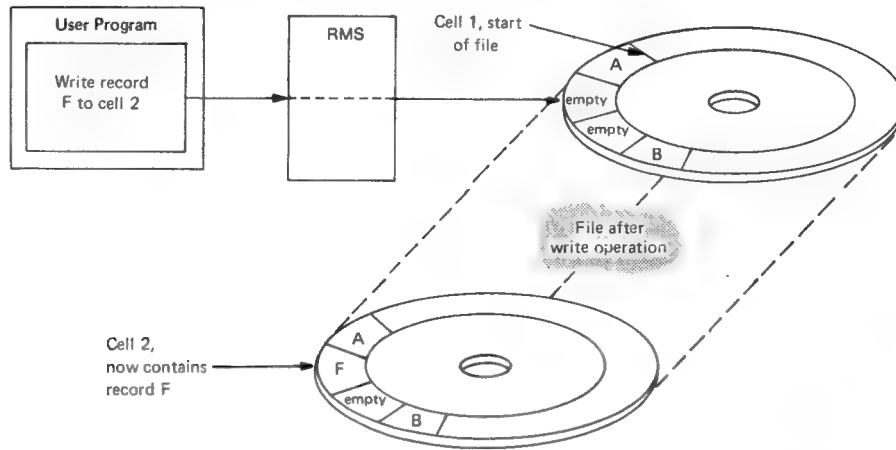
ZK-748-82

When storing records sequentially in a relative file, VAX RMS places each new record in the cell whose relative record number is one higher than the most recently stored cell, provided the cell is empty. If the next cell is not empty, the new record cannot be stored in it. Instead, VAX RMS generates an appropriate message that your application program uses to begin an appropriate error routine.

As Figure 2-3 shows, the program directs VAX RMS to store record F in cell 2. Record A already occupies cell 1 but cell 2 is empty, so VAX RMS can store the record in this cell. If this request is followed by a request to store the next record, VAX RMS would store the record in cell 3, which is also empty. However, if the program tries to store a new record in the next cell which already contains record B, the attempt will fail.

Note that although VAX RMS cannot store a new record in a cell that is already occupied, your program is permitted to modify the record currently occupying the cell.

Figure 2-3 Sequentially Storing Records in a Relative File



ZK-749-82

Sequential Access to Indexed Files

When a program sequentially accesses an indexed file, VAX RMS uses one or more indexes to determine the order in which to process the file records. Because index entries are ordered by key values, an index represents a logical ordering of the records in the file. If you define more than one key for the file, each index associated with a key will represent a different logical ordering of the records in the file. Your program, then, can use the sequential access mode to retrieve records in the logical order represented by any index.

In order to retrieve records sequentially from an indexed file, your program must first specify to VAX RMS a key of reference (for example, primary key, first alternate key, second alternate key, and so on). For succeeding retrievals, VAX RMS uses the appropriate index to retrieve records based on how the records are ordered in the index. If the specified index is being accessed in ascending sort order, VAX RMS returns the record with key value equal to or higher than the key value in the previously accessed record. Conversely, if the specified index is being accessed in descending sort order, VAX RMS accesses the next record having a key value equal to or lower than the key value in the previously accessed record.

In contrast to a request to sequentially retrieve data from an indexed file, a request to sequentially store data in an indexed file does not require a key of reference. Rather, VAX RMS uses the stored definition of the primary key to place the record in the primary index and, where applicable, VAX RMS uses the definition of the appropriate alternate key to place a record pointer in the alternate index. When your program issues a series of requests to sequentially store data, VAX RMS verifies that the key value in each successive record is in the specified order.

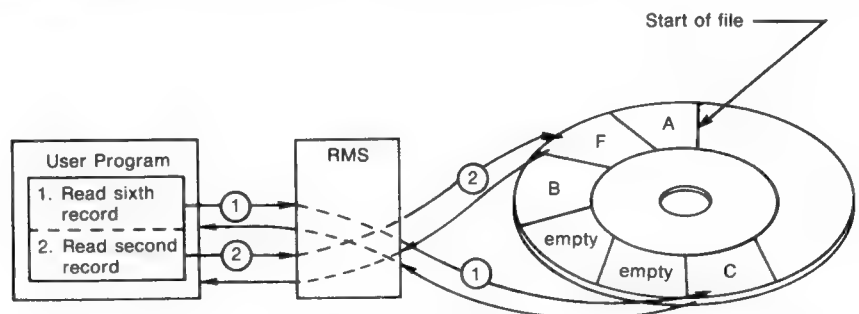
2.1.1.2 Random Access by Key Value or Relative Record Number

Random access is supported for all relative files, all indexed files, and a restricted set of sequential disk files — those having 512-byte, fixed-length records. In random access mode, your program (not the file organization) determines the record processing order. For example, to randomly access a record in a relative file or a record in a sequential disk file having 512-byte, fixed-length records, your program must provide VAX RMS with the relative record number of the cell containing the record. Similarly, to randomly access a record from an indexed file, your program must provide VAX RMS with the appropriate key of reference and key value.

Random Access to Sequential and Relative Files

Unlike sequential access, random access follows no specific pattern. Your program may make successive requests for storing or retrieving records anywhere within the file. In Figure 2-4, the program directs VAX RMS to retrieve the sixth record in a relative file (record C) and then it requests VAX RMS to retrieve record F, the second record.

Figure 2-4 Random Access by Relative Record Number



ZK-750-82

Compare this figure with Figures 2-1 and 2-2.

Random Access to Indexed Files

To randomly access a record from an indexed file, your program must specify both a key value and the index that VAX RMS must search (for example, primary index, first alternate key index, second alternate key index, and so on). When VAX RMS finds a record with a matching key value, it passes the record to your program.

There are several methods your program can use to randomly access a record by key:

- Exact match of key values.
- Approximate match of key values. When accessing an index in ascending sort order, VAX RMS returns the record that has the next higher key value. Conversely, when the index is accessed in descending sort order, VAX RMS returns the record that has the next lower key value.
- Generic match of key values. Generic matching is applicable to string data-type keys only. For a generic match, the program need only specify a match of some specified number of leading characters in the key.
- Combination of approximate and generic match.

Chapter 8 describes these key match conditions in more detail.

In contrast to record retrieval requests, program requests to store records randomly in an indexed file do not require the separate specification of a key value. All keys (primary and any alternate key values) are in the record itself.

When your program opens an indexed file to store a new record, VAX RMS uses the key definitions stored in the file to find each key field in the record and to determine the length of each key. After writing the new record into the file, VAX RMS uses the record's key values to make appropriate entries in the related indexes so that the record can be subsequently accessed using any of its key values.

2.1.1.3 Random Access by Record's File Address

The fact that every record on disk has a unique file address — the record's file address (RFA) — provides another method of randomly retrieving records in all types of file organizations.

Note

RFA mode provides the only means of randomly accessing variable-length records in a sequential file.

An important feature of the RFA is that it remains constant as long as the record is in the file. VAX RMS returns the RFA to your program each time the record is retrieved or stored. Your program can either ignore the RFA or it may keep it as a random-access pointer to the record for subsequent accesses.

Figure 2-5 contains two illustrations. The first shows that when a record is stored in a file, its RFA is returned to the program. The second shows that when the program wants to subsequently access this record randomly, it simply provides VAX RMS with the RFA.

2.1.2 Record Formats

Except for the key values that are part of the records stored in indexed files, VAX RMS is not concerned with record content. Rather, it looks at the record's format, that is, the way the record physically appears on the recording surface of the storage medium.

VAX RMS supports four different record formats:

- Fixed length
- Variable length
- Variable length with fixed-length control
- Stream

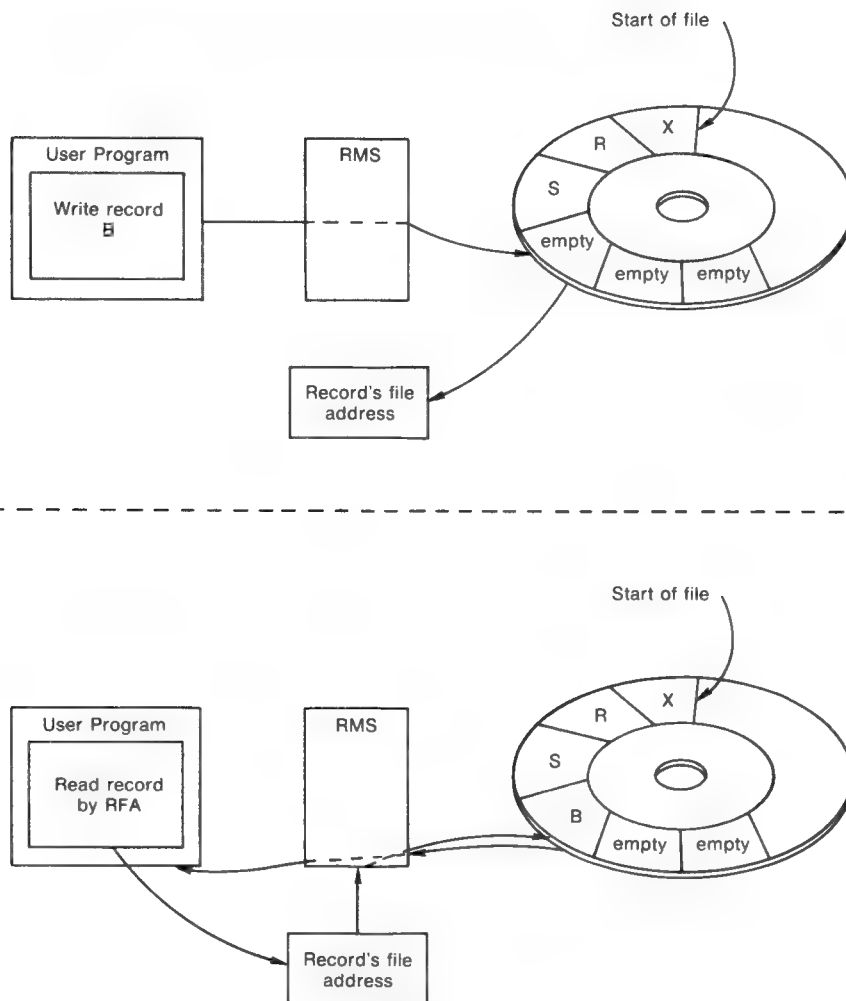
The fixed-length and variable-length record formats are supported for all three file organizations. The variable-length with fixed-length control (VFC) format is supported only for sequential and relative files.

Note

In relative files, all records are stored in fixed-length cells regardless of their format.

The stream format is supported for sequential files only.

Figure 2-5 Random Access by Record's File Address



ZK-751-82

At the VAX MACRO level, you may specify the record format for a file directly by using the `FAB$B_RFM` field in the FAB.

2.1.2.1 Fixed-Length Records

When fixed-length format is specified, all the records in a file are the same length. You set the record length at file-creation time and the specified length becomes part of the information that VAX RMS stores and maintains for the file; it cannot be changed.

For fixed-length format, each record occupies the same amount of space in the file and the specified length must be able to accommodate the longest record in the file. If any record fields are not used, your program must be able to detect them and provide appropriate disposition.

2.1.2.2 Variable-Length Records

When the variable-length record format is specified, each record is only as long as the data within it requires. When VAX RMS stores a variable-length record in a file, it prefixes a *count field* to the record. The count field contains the number of bytes in the record to accommodate retrieval. VAX RMS builds the count field from information in your program and considers the count field separate from the record data fields.

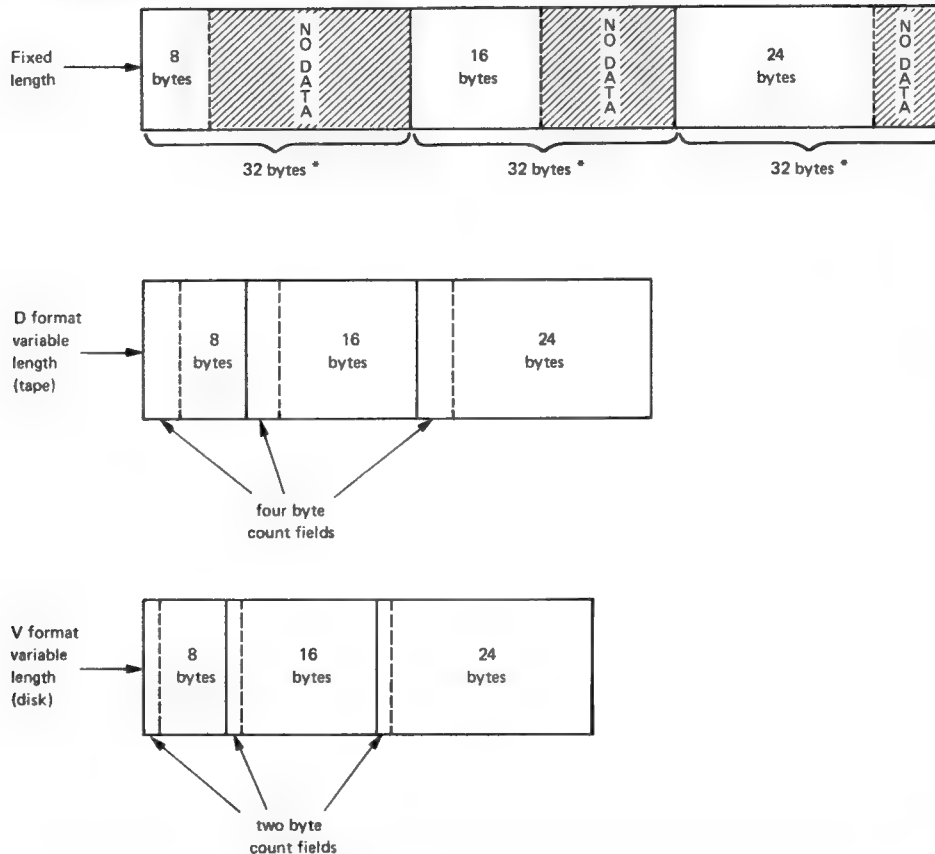
VAX RMS uses two types of variable-length record formats, V format and D format:

- | | |
|----------|--|
| V format | Applies to variable-length records in files residing on disk volumes. VAX RMS prefixes the data portion of each record with a 2-byte binary count field that specifies the length of the record in bytes excluding the byte field itself. |
| D format | Applies to variable-length records residing on tape files. To comply with the American National Standard X3.27-1977, VAX RMS stores a 4-byte decimal count field before the data field of each record on a magnetic tape volume. In contrast to V-format records, the count field is considered as part of the record; but before VAX RMS returns the count, it is adjusted to include only the length of the record data. |

When creating a file of variable-length records, you must specify the value (in bytes) of the largest record permitted in the file. Any attempt to store a record containing more bytes than the specified value results in an error. If you specify a value of 0, any length record can be stored in the file except that the bucket capacity limitation must be observed for relative and indexed files.

Figure 2-6 compares fixed-length and variable-length record formats as they apply to sequential files. Each format shows a portion of a file that contains three records. The comparable record in each format contains the same number of bytes. The first record has 8 bytes, the second, 16, and the third, 24. For the fixed-length format, the record length was set at 32 bytes. Therefore, VAX RMS considers all 32 bytes to be used, when, in fact, unused bytes may exist. Clearly, variable-length records can save space; but if records are to be updated in place, you should consider trading off some space efficiency for update flexibility.

Figure 2-6 Comparison of Fixed- and Variable-Length Records



* RMS considers all 32 bytes to be used, even though they may not contain useful information in the eyes of the user.

ZK-754-82

In the relative file organization, all records, both variable length and fixed length, are stored into fixed-length cells. Here, variable-length records do not save space; in fact, the two count-field bytes prefixing each record actually consume additional space.

In the indexed file organization, variable-length records are truly variable with record length limited only by the capacity of the data bucket or by a defined maximum record size. Because the record size for indexed files may be expanded when the file is updated, there is no requirement to initially specify the maximum record length.

2.1.2.3 Variable-Length with Fixed-Length Control Records

Variable-length with fixed-length control (VFC) records are similar to variable-length records except that a fixed-length control field is prefixed to the variable-length data portion. Unlike variable-length records however, VFC records cannot be used in indexed files.

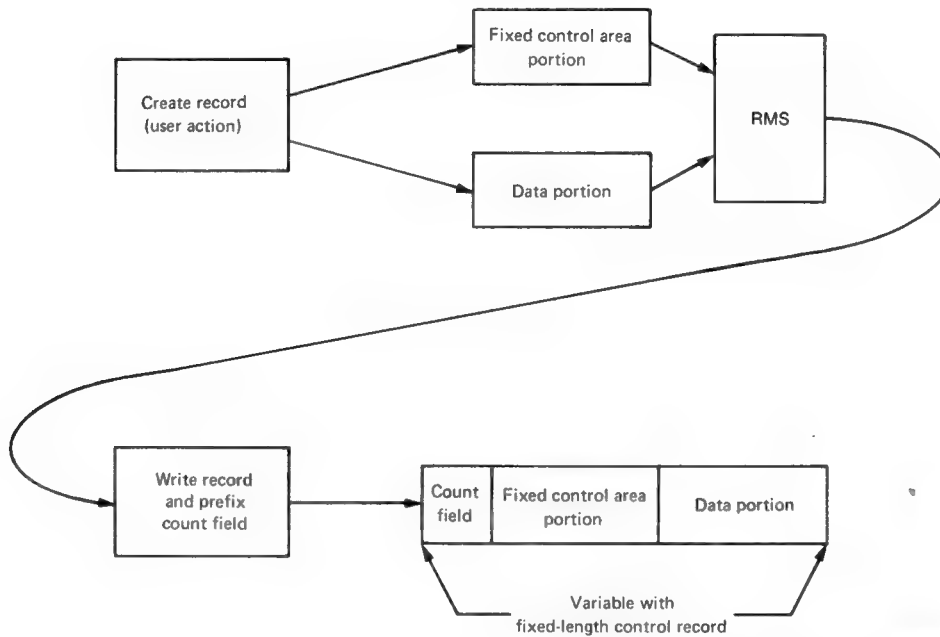
When you create a file for VFC records, you must specify the value (in bytes) of the longest record permitted in the file. Any attempt to store a record containing more bytes than the specified value results in an error. If you specify a value of 0, any length record can be stored in the file except that records in relative files are limited by the capacity of the bucket.

Next, you must specify the value in bytes of the fixed-control area. The fixed-control area allows you to include within the record additional data that may have no direct relationship to the other contents of the record. For example, the fixed-control area may contain line-sequence numbers for every record in the file. The sequence numbers would not be used in any program operations involving the record, but typically could be used to locate records during file editing.

At the VAX MACRO level, you establish the length of the control field for VFC records using the FAB\$B_FSZ field in the FAB. The Open, Create, and Display services provide the control field length as an output in the XAB\$B_HSZ field of the XAB.

When writing a VFC record to a file, VAX RMS merges the fixed control area with the data and then prefixes the record with the count field. Figure 2-7 shows the path VAX RMS takes to store a VFC record to a file.

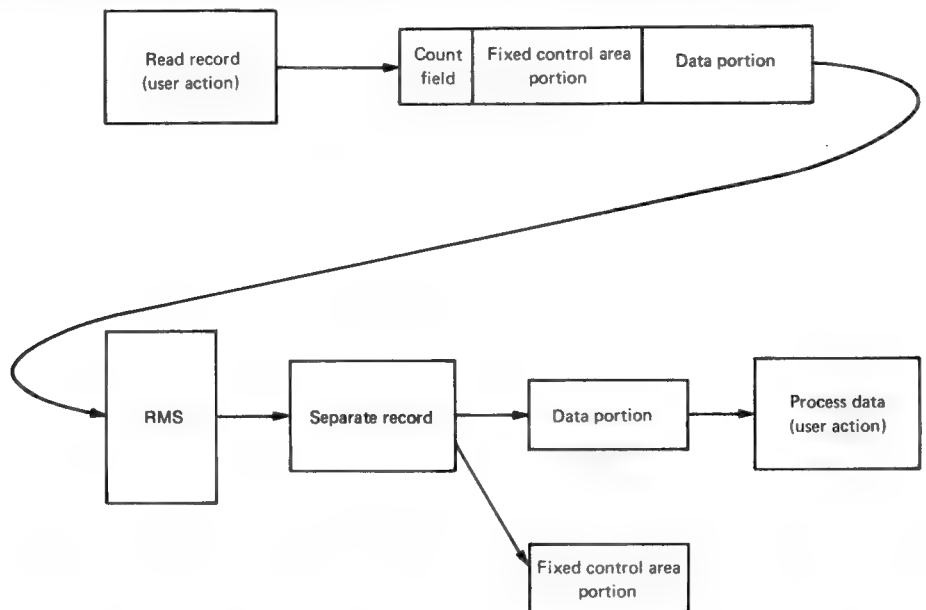
Figure 2-7 Writing a Variable with Fixed-Length Control Record



ZK-755-82

When retrieving a VFC record, VAX RMS uses the count field to determine the overall length of the record and the appropriate file attribute to determine the length of the control area. Next, VAX RMS subtracts the control area length from the overall record length and uses the result to separate the data from the control information. VAX RMS then routes the data to the user program's data processing path and stores the control information in a designated storage area for program use, if applicable. (See Figure 2-8).

Figure 2-8 Retrieving a Variable with Fixed-Length Control Record



ZK-756-82

2.1.2.4 Stream Format

Records in stream format are delimited by special characters or character sequences called *terminators* that are part of the record they delimit. There are three variations of stream format:

- STREAM_ CR With this type of record stream, each record is terminated using a *carriage return* character.
- STREAM_ LF With this type of record stream, each record is terminated using a *line feed* character.
- STREAM With this type of record stream, you have the option of specifying the terminating character from a limited set of special characters, including the carriage return character (CR), the carriage return/line feed character (CR/LF) pair, the form feed (FF) character, or the escape (ESC) character.

In a stream-formatted file, the data is treated as a continuous stream of bytes, without control information such as record counts, segment flags, or other

system-supplied boundaries. VAX RMS supports the stream record format for sequential files on disk devices only.

2.2 File Organization Concepts

The terms *file organization* and *access mode* are closely related, but they are distinct from each other, nonetheless.

You establish the physical arrangement of records in the file — the file organization — when you create it. The organization of a file cannot be changed unless you use a utility conversion routine (such as the Convert Utility) to re-create the file with a different organization.

One of the file attributes you specify prior to creating a file is how records are inserted into it and subsequently retrieved from it — the access mode.

The reason these terms are sometimes confused is because they share common elements. That is, files are *organized* sequentially, relatively, or by keyed-value indexing. Similarly, a file may be *accessed* sequentially, relative to some reference value, or by using a keyed index value. The following descriptions attempt to emphasize the distinctions.

Table 2-2 lists important features of each file organization.

Table 2-2 File Organization Characteristics and Capabilities

Characteristics and Capabilities	Sequential	Relative	Indexed
MEDIUM			
Disk	Yes	Yes	Yes
Magnetic Tape	Yes	No ¹	No ¹
Unit Record	Yes	No	No
RECORD FORMATS			
Fixed-length	Yes	Yes	Yes
Variable-length	Yes	Yes	Yes
VFC (disk only)	Yes	Yes	No
Stream (disk only)	Yes	No	No
Undefined (disk only)	Yes	No	No

¹ Although these file organizations are not compatible with magnetic tape operations, you may use magnetic tape to transport the files.

Table 2-2 (Cont.) File Organization Characteristics and Capabilities

Characteristics and Capabilities	Sequential	Relative	Indexed
OVERHEAD PER RECORD			
	0, 1, or 2 bytes ²	1 or 3 bytes ³	7 to 11 bytes ⁴
RECORD OPERATIONS			
Connect	Yes	Yes	Yes
Delete	No	Yes	Yes
Disconnect	Yes	Yes	Yes
Find	Yes	Yes	Yes
Flush	Yes	Yes	Yes
Free	No	Yes	Yes
Get	Yes	Yes	Yes
Rewind	Yes	Yes	Yes
Truncate	Yes	No	No
Update (disk only)	Yes	Yes	Yes
Put	Yes	Yes	Yes
I/O UNIT			
	1 or more blocks	Bucket	Bucket
I/O TECHNIQUES			
Deferred write	Normal mode	Selectable	Selectable
Multiblock count	Yes	Bucket size	Bucket size
Multiple access streams	Yes	Yes	Yes
Multiple buffers	Yes	Yes	Yes

²Fixed length records and records with undefined format use no overhead; stream format uses either 1 or 2 bytes of overhead; variable-length and VFC records use 2 bytes of overhead.

³Fixed-length overhead is 1 byte; variable-length or VFC, 3 bytes; extra overhead applies to each cell.

⁴For Prolog 1 or 2, the overhead for fixed-length records is 7 bytes and for variable-length records, 9 bytes. For Prolog 3, the uncompressed overhead for fixed-length records is 9 bytes and for variable-length records, 11 bytes. For key compression, add 2 bytes.

Table 2-2 (Cont.) File Organization Characteristics and Capabilities

Characteristics and Capabilities	Sequential	Relative	Indexed
I/O TECHNIQUES			
Mass insertion	No	No	No
Access sharing	Read/write	Read/write	Read/write
Other features	Block-spanning records	Maximum record number	Areas

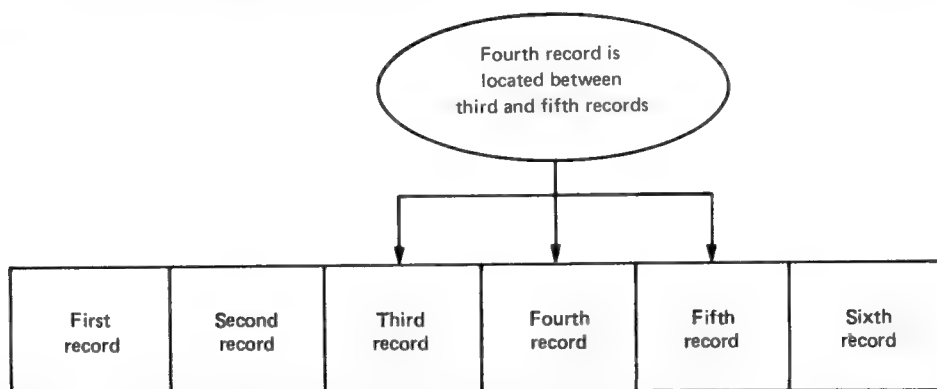
The next three sections describe the various file organizations in detail.

2.2.1 Sequential File Organization

The sequential file organization is supported for all device types; indeed, for all devices except disk devices, it is the only organization supported.

In sequential file organization, records are arranged one after the other in the order in which they are stored in the file. For example, the fourth record is located between the third and fifth records, as illustrated in Figure 2-9.

Figure 2-9 Sequential File Organization



ZK-742-82

Choosing A File Organization

You cannot insert new records between existing records because no physical space separates them. Therefore, you can only add records to the current end of the file, that is, immediately following the most previously stored record. For the same reason, you cannot add to the length of an existing record when updating it.

The advantages and disadvantages of the sequential file organization are listed in Table 2-3.

Table 2-3 Sequential File Organization Advantages and Disadvantages

Advantages	Disadvantages
Simplest organization	To get a particular record, most higher-level languages must access all the records prior to it — no random access by key ¹
Minimum overhead on disk	Interactive processing is awkward; operator must wait as the program searches for a record
Allows block spanning	Certain compiled programs cannot access a record already passed without closing and re-opening the file
Optimal if application accesses all records on each run	You can add records only at end of file ¹
Most versatile in formats: Exchange data with systems other than VAX RMS; compatible with ANSI magnetic tape format	
No restrictions on the type of storage media; the file is portable	

¹These restrictions do not exist for disk sequential files with fixed-length record format; records in such files can be stored and retrieved using random by key access, depending on your higher-level language capabilities.

Table 2-3 (Cont.) Sequential File Organization Advantages and Disadvantages

Advantages	Disadvantages
Random-by-key record access available on fixed-format disk sequential files	

2.2.2 Relative File Organization

The relative file organization allows sequential and random access of records but is supported on disk devices only. In fact, relative files provide the fastest random access, and require fewer tuning considerations.

A relative file consists of a series of fixed-length record positions (or cells) numbered consecutively from 1 to n that enables VAX RMS to calculate the record's physical position on the disk. The number, referred to as the *relative record number*, indicates the record cell's position relative to the beginning of the file.

VAX RMS uses the relative record number as the key value to randomly access records in a relative file. A good way to keep track of relative record numbers is to assign them based on some numeric field within the record; for example, the account number.

See Section 2.1.1.2 for a detailed description of random access by key.

Each record in the file may be randomly assigned to a specific cell. For example, the first record may be assigned the first cell and the second record may be assigned the fourth cell, leaving the second cell and third cell empty. Unused cells and cells from which records have been deleted may be used to store new records.

Figure 2-10 illustrates the relative file organization.

In a relative file, the actual length of the individual records may vary (that is, different size records can be in the same file) subject to the limits imposed by the specified cell length. For example, assume a relative file having a specified cell length of 256 bytes and three records where the first record is 192 bytes long, the second is 128 bytes, and the third is 253 bytes. The example file is depicted in Figure 2-11.

Figure 2-10 Relative File Organization

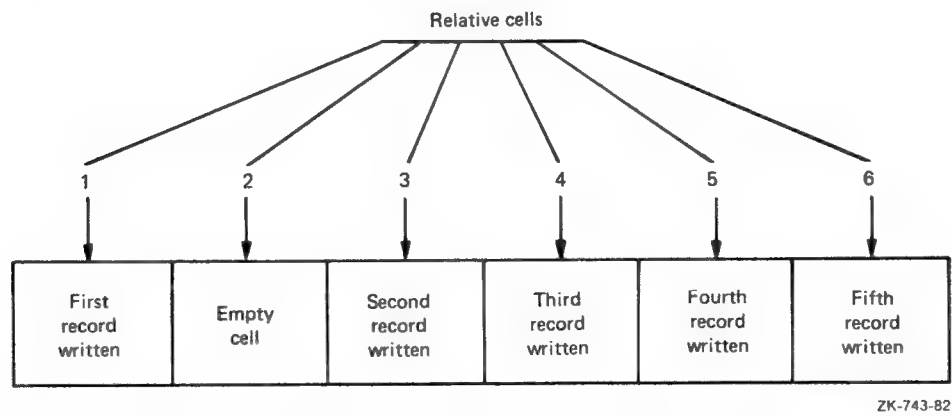
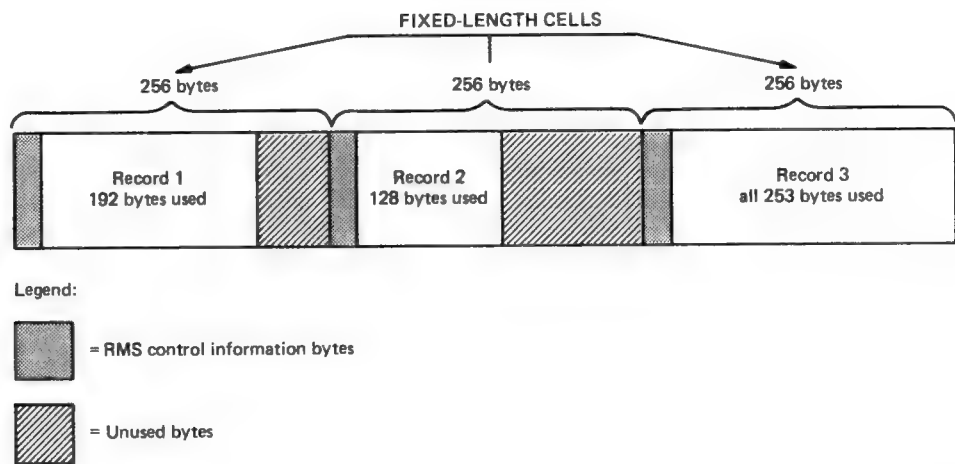


Figure 2-11 Variable-Length Records in Fixed-Length Cells



Note that because the records are variable-length, each is prefixed by 3 bytes, the 2-byte count field described in Section 2.1.2.2 and a 1-byte field that

indicates whether or not the cell is empty (a delete flag). These bytes are used only by VAX RMS and you need not be concerned with them, except when planning the file's space requirements.

The advantages and disadvantages of the relative file organization are listed in Table 2-4.

Table 2-4 Relative File Organization Advantages and Disadvantages

Advantages	Disadvantages
Random access in all languages	Restricted to disk devices
Allows deletions	File contains a cell for each cell number between first and last record in file; limits data density
Allows random Get and Put operations	Program must know relative record number or RFA of record before it can randomly access the data; no generic access as in indexed file organization
Random and sequential access with low overhead	Interactive access can be awkward if you do not access records by relative record number
Can be write-shared	You can only insert records into unused record cells, but you can update existing records
	VAX RMS does not allow duplicate relative record numbers
	The space taken up by each record is as long as the longest record in the file

2.2.3 Indexed File Organization

The indexed file organization allows sequential and random access of records but is supported on disk devices only. This type of file organization lets you store data records in an index structure ordered by the primary key and to retrieve data using index structures ordered by alternate keys. The

alternate index structures do not contain data records; instead, they contain appropriate pointers to the data records in the primary index.

For example, an indexed file may be ordered in ascending sort order by the primary key "employee number." However, you may want to set up additional (alternate) indexes for retrieving records from the file. Typically, you might set up an alternate index that is ordered in descending sort order by each employee's social security number.

Note

The physical location of records in an indexed file is transparent to your program because VAX RMS controls record placement.

In addition to the indexes, each indexed file includes a prologue structure that contains information about the file, including file attributes. VAX RMS currently supports three distinct prologues — Prolog 1, Prolog 2 and Prolog 3 — but VAX RMS will normally create a Prolog 3 indexed file. However, you can specify a previous prologue version, typically for compatibility with RMS-11.

2.2.3.1 Sequentially Retrieving Indexed Records

To sequentially retrieve indexed records, your program must specify the key for the first access. VAX RMS then uses the index for that key to retrieve successive records. For example, assume there is an index file with three records, having primary keys of A, B, and C, respectively. To retrieve these records sequentially in ascending sort order, your program must provide the key A on the first access; VAX RMS will access the next two records without further key inputs from your program.

To randomly retrieve records in an index file, your program must provide the appropriate key value for each access. Now assume an index file with three records having primary keys A, B, and C that will be retrieved in C, A, B order. On the first access, your program must provide the key C, on the next access the key A, and on the final access the key B.

2.2.3.2 Index Keys

In an indexed file, each record includes one or more key fields (or simply keys) that VAX RMS uses to build related indexes. Each key is identified by its location within the record, its length and whether it is a simple key or a segmented key.

A simple key may be any one of the following data types:

- A single contiguous character string
- A packed decimal number
- A 2-, 4-, or 8-byte unsigned binary number
- A 2-, 4-, or 8-byte signed integer.

Note

RMS-11 cannot process 8-byte numeric keys.

Segmented keys are fields of character strings having from 2 to 8 segments which may be or may not be contiguous; however, VAX RMS treats all key segments as a logically contiguous string.

For an indexed file, you must define at least one key, the primary key, and you may optionally define one or more alternate keys. VAX RMS uses alternate keys to build indexes that identify records in alternate sort orders. As with the primary key, each alternate key is defined by its location and length within the record.

2.2.3.3 Other Key Characteristics

In addition to defining keys, you can specify various key characteristics (FDL secondary key attributes) including the following:

Choosing A File Organization

Duplicate keys	When you specify this characteristic, you may use the key value in more than one record. However, only the first record having the key value can be accessed randomly; other records having the same key value can only be accessed sequentially.
Changeable keys	This characteristic applies to alternate keys only. When you specify <i>changeable</i> alternate keys, the <i>alternate</i> keys in a record can be changed when the record is updated. When an alternate key value is changed, VAX RMS automatically adjusts the appropriate index to reflect the new key value.
Null keys	This characteristic applies to alternate keys only. When you insert a null in an alternate key field, the record is effectively removed from the related index.

Note

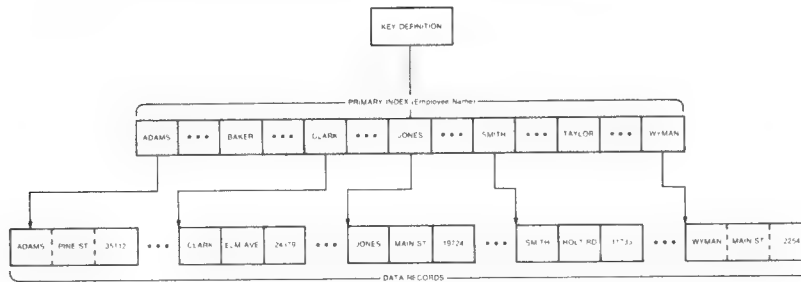
Any record that is not long enough to contain a complete alternate key field is also removed from the related index.

You may nullify any key characteristic specified by default. For example, if a key can be used in more than one record by default (duplicate characteristic), you can nullify the characteristic so that the key value cannot be used in more than one record. The same rule applies to alternate key values that are changeable by default, or alternate keys that can be assigned a null value by default.

When you disallow duplicate key values, VAX RMS rejects any attempt to put a record into a file if it contains a key value that duplicates a key value already present in another record. Similarly, when alternate key values cannot be changed, VAX RMS does not allow your program to update a record by changing the alternate key value. If you disallow a null value for a key, VAX RMS inserts an entry for the record in the associated alternate index.

Figure 2-12 illustrates the general structure of an indexed file that has only the primary key defined, the primary key being the names of employees in an employment record file. Figure 2-13 illustrates the general structure of an indexed file that has the primary key and one alternate key defined. The primary key is the names of employees, and the alternate key is the badge number of employees in an employment record file.

Figure 2-12 Single-Key Indexed File Organization



2.2.3.4 Specifying Sort Order

VAX/VMS Version 4.4 lets you specify either ascending sort order or descending sort order for each key. At the VAX MACRO level, you encode sort order within the key data type field (XAB\$B_DTP) of the associated key XAB; you use the attribute KEY TYPE at the FDL level. For example, if you want to build an index of string data type keys in ascending sort order using VAX MACRO, you enter the following line in the associated key XAB:

DTP = STG

If instead you want to build an index of string data type keys in descending sort order, you enter this line in the associated key XAB:

DTP = DSTG

See the *VAX Record Management Services Reference Manual* for a complete listing of key data types used to specify ascending and descending sort order.

The advantages and disadvantages of the indexed file organization are listed in Table 2-5.

Figure 2-13 Multiple-Key Indexed File Organization

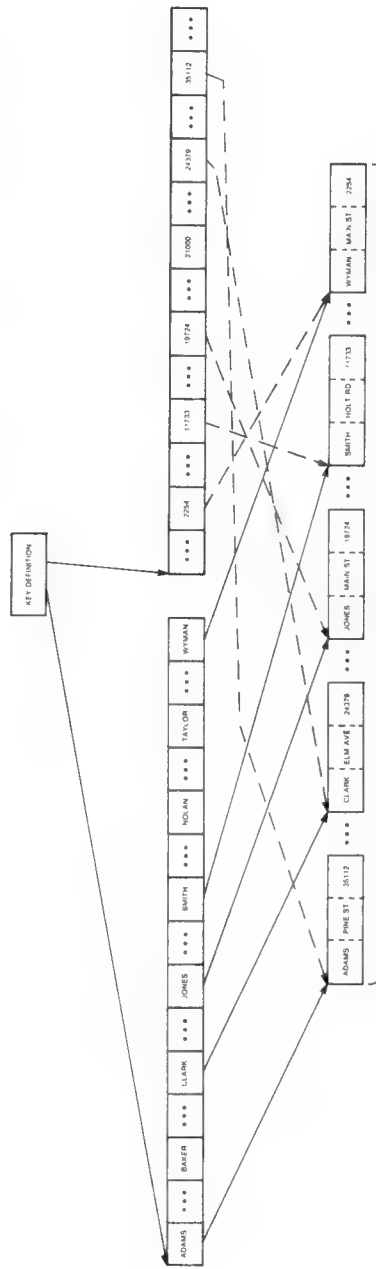


Table 2-5 Indexed File Organization Advantages and Disadvantages

Advantages	Disadvantages
Most flexible random access: by any one of multiple keys or RFA; key access by generic or approximate value	Highest overhead on disk and in memory
Duplicate key values possible	Restricted to disk
Automatic sort of records by primary and alternate keys; available during sequential access	Most complex programming
Record location is transparent to user	Longest record access times
Potential range of key values not physically present as in relative file organization	
Variety of data formats for keys	

3

Performance Considerations

When you design a file, your decisions regarding record access mode, record format, and file organization should be aimed at achieving optimum data processing performance for your application. This chapter discusses general performance considerations and specific trade-offs you can make in the design of your data files. In Sections 3.3, 3.4, and 3.5, these trade-offs are discussed in the contexts of the three file organizations.

3.1 Design Considerations

In designing files for optimum data processing performance, you should put particular emphasis on the following performance factors:

- **Speed** — You want to increase the speed with which your program processes data.
- **Space** — You want to decrease the space required to store data on disk and to process data in memory.
- **Shared access** — You want your data to be accessible to others who are authorized access to it.
- **Impact on application design** — You should design files that require minimum application design effort.

3.1.1 Speed

The first guideline you can apply to the design process is to decrease I/O time.

Storing data on, and retrieving data from, mass storage devices is the most time-consuming VAX RMS operation. For example, when an application needs data, the disk controller must first search for the data on the disk. The disk controller must then transfer the data from the disk to main memory. After processing the data, the program must provide for returning the results to mass storage via the I/O subsystem.

One way to reduce I/O time is to have the data in memory so that you can minimize search and transfer operations. If data must be transferred to memory for processing, you should consider design variables that reduce transfer time.

The first variable you might consider is the set of *file attributes* that may affect I/O time:

- Initial file allocation
- Default extension quantity
- Bucket size (for a relative or indexed file)
- Number of keys (for an indexed file)
- Number of duplicate key values (for an indexed file)

The second variable is the *file size* as measured by the number of records in the file. File size affects the time it takes to scan a file sequentially or to access records using an index.

The final variable is the *storage device* on which your program and data files reside. The type of device chosen (moving-head, fixed-head, and so on) and the amount of I/O activity for that device within the system are crucial to I/O performance.

To make your applications run faster, consider the following:

- Keep as much data in memory as possible, but be wary of any significant increase in the page fault rate.
- Minimize the number of I/O transfers by transferring larger portions of data.
- Arrange your data on the disk to minimize disk head motion.

3.1.2 Space

When you run your application, you need space to buffer data in memory. You can speed up data processing by increasing the size of the I/O buffers VAX RMS uses, but avoid exceeding the space limitations imposed by the working set.

In addition to the data buffers themselves, the space required to store data can vary depending on the file organization you choose.

For example, sequential file organization requires VAX RMS to add an empty byte to a record when the record has an odd number of bytes but must be aligned on an even-numbered byte boundary. At the record level, you should consider the added space required to prefix a 2-byte count field to each variable-length record.

For the relative file organization, VAX RMS constructs a series of record storage cells based on the length of the records. The record cells are 1 byte longer than the size of fixed-length records or 3 bytes longer than the maximum size specified for variable-length records.

For the indexed sequential file organization, VAX RMS must add the following informational components to your data files:

- An index for each defined key
- 15 bytes of formatting information for each bucket
- A 7-byte header for each record
- A count field for each variable-length record
- Other overhead of varying lengths that is needed by VAX RMS to move files and to delete records. You should keep the size of records to the minimum required for your application.

You should also consider the effects of compression on the size of your indexed files. You can compress keys in data buckets and in index buckets, and you can compress data in the primary buckets. If you use key, index, or data compression, the file requires less space on the disk, and each I/O buffer can hold more information. Compression may even eliminate one index level thereby reducing the number of disk transfers needed for random access.

Random access of compressed files requires slightly more CPU time, but this is usually offset by the improved performance you achieve with fewer index levels.

3.1.3 Shared Access

A file management technique that allows more than one user to simultaneously access a file or a group of files is called *shared access* or *file sharing*. When you try to adjust the performance of shared files, you need to pay particular attention to record locking options and the use of global buffers. Avoid assigning sharing attributes to files that are not actually shared.

There are essentially three sharing conditions: no sharing, sharing without interlocking, and sharing with interlocking. Chapter 7 discusses each of these in detail.

3.1.4 Impact on Applications Design

The impact on applications design increases as file design complexity increases. That is, your application programs require more design effort for processing indexed files than for processing sequential files. The primary consideration here should be to evaluate whether the benefits derived by having direct access to records is worth the added cost of the application program design needed to interface with the file management system.

3.2 Tuning

The process of designing your files to achieve better processing performance is called *tuning*.

Tuning requires you to make a number of trade-offs and design decisions. For example, if a process had sole access to the processor, it could keep all of its data in memory and tuning would be unnecessary; but this situation is highly unlikely. Instead, several processes are usually running simultaneously and are vying for the memory resource. If all processes demand large amounts of memory, the system will respond by paging and swapping, which slows down the system performance.

The way you intend to use your programs and data files can determine some of the basic tuning decisions. For example, if you know that three files are accessed 80 percent of the time, you might consider locating the files in a common area on the disk to speed up access to them. The performance of programs that use the other files will be slower, but the system as a whole will run faster.

In tuning your file management system, you implement these trade-offs and design decisions by specifying file design attributes together with various file-processing options and record-processing options.

3.2.1 File Design Attributes

The following file design attributes control how the file is arranged on the disk and how much of the file is transferred to main memory when needed. These file design attributes apply generally to all three types of file organization; other file design attributes that pertain specifically to the various file organizations are described under the appropriate heading.

- Initial file allocation
- Contiguity
- File extend quantity
- Units of input/output

- The use of multiple areas (for indexed files)
- Bucket fill factor (for indexed files)

The following sections discuss how each file design attribute can maximize efficiency.

3.2.1.1 Initial File Allocation

When you create a file, you should allocate enough space to store it in one contiguous section of the disk. If the file is contiguous on the disk, it requires only one retrieval pointer in the header; this reduces disk head motion.

You should also consider allocating additional space in anticipation of file growth to reduce the number of required extensions.

You can set this value either by using the FDL attribute FILE ALLOCATION or by using the file access control block field FAB\$L_ALQ.

3.2.1.2 Contiguity

Use the FILE secondary attribute CONTIGUOUS to arrange the file contiguously on the disk, if you have sufficient space. If you assign the file the CONTIGUOUS attribute and there is not enough contiguous space on the disk, VAX RMS will not create the file.

A better choice might be to use the FDL attribute BEST_TRY_CONTIGUOUS. This attribute arranges the file contiguously on the disk if there is sufficient space or non-contiguously if the space is not available for a contiguous file.

You can make this choice by taking the FDL default values for both attributes — NO for CONTIGUOUS, YES for BEST_TRY_CONTIGUOUS or by taking the VAX RMS option FAB\$V_CBT in the FAB\$L_FOP field.

3.2.1.3 Extending a File

VAX RMS automatically extends files when more space is needed than originally allocated. You have the option of specifying a default extension value (in blocks) for a file or having VAX RMS automatically specify the extension value.

If you expect to add large amounts of data to a file over a relatively short time period, you should consider specifying an extension value that is large enough to minimize file fragmentation and to reduce the total overhead consumed by EXTEND operations. As a file becomes fragmented, access

time becomes greater and processing performance degrades. Similarly, added overhead due to VAX RMS having to extend files can degrade performance.

Conversely, if you only add small amounts of data to the file over a relatively long period, specifying a large extension value would result in wasted disk space.

If you do not set a default extension quantity, VAX RMS will automatically extend the file when necessary using an algorithm that allocates a minimal extension value, writes the data to the file, and then truncates any unused space when it closes the file. When a file becomes filled, this technique can result in severe file fragmentation.

There are several ways you can specify the default extension value:

- When you create the file, you can have the FDL editor (EDIT/FDL) calculate the value using information you provide via the script. This is the recommended method.
The FDL editor assigns the value using the FILE EXTENSION attribute. For index files having multiple areas, the FDL editor assigns a separate value to each area using appropriate AREA EXTENSION attributes.
- If you choose not to use FDL to establish the extension value when you create the file, you can specify the value directly in the FAB field FAB\$W_DEQ if you are using a low-level language. For index files having multiple areas, you can assign separate values to each area using appropriate XAB\$B_AID fields and related XAB\$W_DEQ fields. Here you must determine the value using the average record size, the number of records that will be added to the file during a given period of time, the number of records per bucket, and the bucket size.
- If you elect not to assign an extension value using FDL or by using FAB's and XAB's directly, you can establish the extension value at run-time using the DCL command SET FILE/EXTENSION=*n* where *n* is the default extension size the volume.
- If you decide not to use any of the previously described methods, you can use the value established by the DCL command SET RMS_DEFAULT/EXTEND_QUANTITY=*n*, where *n* is the extension value.
This command applies the VAX RMS default extension value to files for your process only unless you use the /SYSTEM qualifier with the command. Note that this requires the CMKRNL privilege.

Note

If you set the default extension size with any other method, the value you specify with the SET RMS_DEFAULT command will be overridden.

3.2.1.4 Units of Input/Output

Another file design strategy is to reduce the number of times that VAX RMS must transfer data from disk to memory by making the I/O units as large as possible. Each time VAX RMS fetches data from the disk, it stores the data in an I/O memory buffer whose capacity is equal to the size of one I/O unit. Having a larger I/O unit will make more records immediately accessible to your program from the I/O buffers.

In general, using larger units of I/O for disk transfers improves performance, as long as the data does not have to be swapped out too frequently. However, as the total space used for I/O buffers in the system approaches a point that results in excessive paging and swapping, increasing I/O unit size will actually degrade system performance.

VAX RMS performs I/O operations using one of the following I/O unit types:

- Blocks
- Multiblocks
- Buckets

A *block* is the basic unit of disk I/O and it consists of 512 contiguous bytes. Even if your program requests less than a block of data, VAX RMS automatically transfers an entire block. The other I/O units — multiblocks and buckets — are made up of block multiples. A *multiblock* is an I/O unit that includes up to 127 blocks but whose use is restricted to files that are organized *sequentially*. See Section 3.3.2 for details.

A *bucket* is the I/O unit for relative and indexed files and it may include up to 63 blocks. See Sections 3.4 and 3.5 for details.

3.2.1.5 Multiple Areas for Indexed Files

For indexed files, another design strategy is to separate the file into multiple *areas* where each area has its own extension size, initial allocation size, contiguity options, and bucket size. You can minimize access times by precisely positioning each area on a specific disk volume, cylinder, or block.

For instance, you can place the data area on one volume of a volume set and place the indexed area on another volume of the volume set. If your application is I/O bound, this strategy could increase its throughput. You can also ensure that your data buckets are contiguous for sequential access by allocating extra space to the data area for future extensions.

3.2.1.6 Bucket Fill Factor for Indexed Files

Any attempt to insert a record in an indexed file that results in a bucket overflow causes a *bucket split*. When a bucket split occurs, VAX RMS tries to keep half of the records (including the new record if applicable) in the original bucket and moves the remaining records to a newly created bucket.

Excessive bucket splitting can degrade system performance through wasted space, excessive processing overhead and file fragmentation. For example, each record that moves to the new bucket must maintain a pointer in the original bucket that indicates the record's new location. At the new bucket, the record must maintain a backward pointer to its original bucket in order to provide a means for updating the pointer in the original bucket when the record moves again.

The pointer in the original bucket must be maintained because each record is accessed through its original bucket. That is, when a program attempts to access a record, it must first go to the bucket where the record originally resided, read the pointer to the record's current bucket residence, and then finally access the record in its current bucket.

To avoid bucket splits, you should permit buckets to be only partially filled when records are initially loaded. This will provide your application with space to make additional random inserts without overfilling the affected bucket.

Section 3.5.2.2 describes fill factors in more detail.

3.2.2 Processing Options

Five processing options can be used to improve I/O operations; 2 file-processing options and 3 record-processing options. The file-processing options include the global buffer option and the deferred-write option. The global buffer option may be used with all 3 file organizations but the deferred write option is restricted to use with relative and indexed files.

The record-processing options include the multiple buffer option, the read-ahead option and the write-behind option. The multiple buffer option may be used with all 3 file organizations but the read-ahead option and the write-behind option may only be used with sequential files.

This section only summarizes the various options. Sections 3.3 through 3.5 describe these options in the context of tuning files. See Chapter 7 for additional information on the use of buffering.

3.2.2.1 Multiple Buffers

When a program accesses a data file, it brings the file from disk into memory using I/O units of blocks, multiblocks, or buckets. The I/O units are subsequently placed in memory *I/O buffers* sized to be compatible with the I/O units.

If you do not have enough buffers, excessive I/O transfers may degrade the performance of your application. On the other hand, if you have too many buffers, performance may be degraded by an overly large working set. As a general rule, however, increasing the size and number of buffers can improve performance if the data read into the buffers will soon be processed and if your working set can efficiently maintain the buffers. In fact, changing the size and number of buffers is the quickest way to improve the performance of your application when you are processing localized data.

The optimum number of buffers depends on the organization and use of your data files. The recommended way to determine the optimum number of buffers for your application is to use the Edit/FDL Utility.

The number of I/O buffers is a run-time parameter which you can set with the FDL editor by adding the CONNECT secondary attribute MULTIBUFFER_COUNT (see Chapter 9) to the definition file. If you are using a low-level language, you can set the value directly into the RAB\$B_MBF field of the record access block for use following connection to the record stream.

Alternatively, the number of buffers may be specified using the DCL command SET RMS_DEFAULT/BUFFER_COUNT=*n*, where the variable *n* represents the desired number of buffers. With this command, you may set distinct values for your sequential, relative and indexed files using the appropriate file organization qualifier. If you omit the file organization qualifier, sequential organization is assumed.

Note

To examine the current settings for the multibuffer values, use the DCL command SHOW RMS_DEFAULT.

If none of the above methods is used, VAX RMS uses the system-wide default value established by the system manager. If the system-wide default is omitted or is set to 0, VAX RMS uses a value of 1 for sequential and relative files and a value of 2 for indexed files.

Refer to Section 3.3.3 for more details on using multiple buffers with sequential files, to Section 3.4.2 for more details on using multiple buffers with relative files and to Section 3.5.2.3 for more details on using multiple buffers with indexed files. Chapter 7 describes the use of multiple buffers in the context of shared files.

3.2.3 Global Buffers

If several processes are to share a file, you may want to provide the file with *global buffers* — I/O buffers that two or more processes can access. With global buffers, processes may access file information without allocating dedicated buffers thereby conserving buffer space and buffer management overhead. You gain this benefit at the cost of additional system resources, as described in the *VAX Record Management Services Reference Manual*.

When you create a file, you can assign it the desired number of global buffers by using the FDL editor to set the value in the FILE secondary attribute GLOBAL_BUFFER_COUNT. From a low-level language, you can optionally set the value directly into the FAB\$W_GBC field. Alternatively, you may use the DCL command SET FILE/GLOBAL_BUFFERS to set the global buffer count.

Sections 3.3 through 3.5 discuss the use of global buffers in tuning the various file types.

3.2.4 Deferred Write Processing

One way to improve performance through minimized I/O is to keep data in memory as long as practicable using the deferred-write option. However, you must analyze your application to determine if this added performance benefit is worth the increased risk of losing data if the system crashes before a buffer is transferred to disk.

With indexed files and relative files, you may use the *deferred write* option to defer writing modified buckets to disk until the buffer is needed for another purpose or until the file is closed. In this way, a cached buffer may be accessed and manipulated without having to use a disk transfer.

Typically, the largest gains in performance come from using the deferred-write option with sequential access. Retrieving and modifying successive records means that you will access all of the records from one bucket while the bucket is in memory.

You may also use this option to prevent VAX RMS from writing a shared sequential file to disk on each modification, thereby improving performance. Again, this increases the risk of losing data if the system crashes before the full buffer is transferred to disk.

Note that sequential files that are not shared behave much as if the deferred-write option were always specified, because a buffer is only written to disk after it is completely filled.

Deferred write is a default run-time option for some high-level languages and can be specified by clauses in other languages. You can explicitly activate this run-time option by using the FDL editor to add the CONNECT secondary attribute DEFERRED_WRITE. If you are using a low-level language, you activate this feature by setting the FAB\$V_DFW bit in the FAB\$L_FOP field of the file access block.

3.2.5 Read-Ahead and Write-Behind Processing

The operation of sequentially organized files can be improved by implementing read-ahead and write-behind processing. These features improve performance by permitting record processing and I/O operation to occur simultaneously. The read-ahead and write-behind features are default run-time attributes with some languages but must be explicitly specified by others.

These features are implemented by using two buffers with the record stream; one being accessed by the processing program, and the other being accessed by the I/O subsystem. For read-ahead, the program reads data from one buffer as the second buffer is inputting more data from disk. For write-behind, one buffer accepts output data from the program, while the second buffer transfers data to disk that was previously output by the program.

The next section provides additional information on read-ahead and write-behind.

3.3 Tuning a Sequential File

Sequential files are relatively simple in construction consisting of a file header and a series of data records. Records are stored in the order in which they are written to the file and may be in any one of the following formats:

- fixed
- variable
- variable with fixed-control length (VFC)
- stream
- undefined

In the fixed-length format, the record length is specified to accommodate the record having the most data. Any record that has less data effectively wastes a corresponding amount of space. If records are permitted to cross block boundaries (block spanning feature), the maximum record length for the fixed format in a sequential file is 32,765 bytes; otherwise the maximum length is limited to 512 bytes (a block).

Although variable-length records can be of different lengths, you may specify the maximum length for a file to VAX RMS. Each variable-length record begins with a 2-byte byte count field that VAX RMS uses when retrieving the record. As with the fixed length format, the maximum record length is 32,765 bytes if records are permitted to cross block boundaries, otherwise the maximum length is limited to 512 bytes.

VFC records consist of a fixed control area having up to 255 bytes, and a variable data area. The control area can be used for any purpose, but it is commonly used to store information such as line numbers or the carriage control in PRINT files. The maximum size of a record with block spanning is 32,763 bytes less the number of bytes in the fixed control area.

There are three variations of the stream format:

- STREAM
- STREAM_LF
- STREAM_CR

All three variations feature a continuous stream of bytes but are different regarding the manner in which records are delimited. The delimiter in the STREAM variation is either a line feed (LF), a vertical tab (VT or CTRL/K), a form feed (FF), or the default combination of carriage return-line feed (CR LF). The delimiter in the STREAM_CR variation is the carriage return (CR) character and the delimiter in the STREAM_LF variation is the line feed (LF) character.

Records are limited to 32,767 bytes in length.

Undefined records are a continuous stream of bytes with no specified terminator. The following sections provide guidelines for improving the performance of sequential file processing using various tuning options.

3.3.1 Block Spanning

You should always specify that records in a sequential file are permitted to span blocks, that is, cross block boundaries. In this way, VAX RMS can pack the records efficiently and avoid wasted space at the end of a block.

By default, the FDL editor activates block spanning for files organized *sequentially* by setting the RECORD secondary attribute BLOCK_SPAN to YES. If you are using a low-level language, you activate block spanning directly in the file access block by setting the FAB\$V_BLK bit in the FAB\$L_RAT field.

3.3.2 Multiblock Size

A *multiblock* is an I/O unit that includes up to 127 blocks but whose use is restricted to files having sequential organization. When a program instructs VAX RMS to fetch data within a multiblock, the entire multiblock is copied from disk to memory.

You specify the number of blocks in a multiblock using the *multiblock count*, a run-time attribute. If you are using the FDL editor, specify the count using the secondary CONNECT attribute, MULTIBLOCK_COUNT. From a lower-level language, you may set the value into the RAB\$B_MBC field, directly. Another alternative is to establish the count using the following DCL command:

```
$ SET RMS_DEFAULT/BLOCK_COUNT=n
```

where the variable *n* represents the specified number of blocks. Here, the specified multiblock count is limited to your process unless you specify the /SYSTEM qualifier.

To see the current VAX RMS default multiblock count, give the DCL command SHOW RMS_DEFAULT.

In most cases, the largest practical multiblock value to specify is the number of blocks in one track of the disk, a number that varies with the various types of disks. (See the *VAX/VMS I/O Reference Volume* for the track sizes in blocks of DIGITAL-supported disks). However, the most efficient number of blocks for your application may be more or less than the number of blocks in a track and you should try various sizes of multiblocks until you find the optimum value.

3.3.3 Number of Buffers

For sequential files, you can specify the number of buffers at run time. From FDL, you can set the number of buffers with the secondary CONNECT attribute MULTIBUFFER_COUNT and from a low level language you can set the value directly into the VAX RMS control block field RAB\$B_MBF. You can set also the number of buffers with the DCL command SET RMS_DEFAULT/SEQUENTIAL/DISK/BUFFER_COUNT=*n*

In simple operations with sequential files, one I/O buffer is sufficient. In fact, increasing the number of buffers takes space in the process working set and could actually degrade performance.

With sequential files, particularly if you want to perform sequential access, you can use read-ahead and write-behind processing. With this type of processing, two buffers are used alternately; one buffer contains the next records to be read or written to the disk while the other buffer completes I/O.

The length of the buffers used for sequential files is determined by the specified multiblock count. The optimal number of blocks per buffer depends on the record size for sequential access to a sequential file, but a value such as 16 may be appropriate.

To see what the current default buffer count is, give the DCL command `SHOW RMS_DEFAULT`. To set the default buffer count, use the DCL command `SET RMS_DEFAULT/SEQUENTIAL/BUFFER_COUNT=n`, where `n` is the number of buffers.

3.3.4 Global Buffers

If a file is shareable, you may want to allocate global buffers to it. A *global buffer* is an I/O buffer that two or more processes can access. If two or more processes are requesting the same information from a file, I/O can be minimized because the data is already in the global buffer. This is especially true for program sequences where all of the processes are reading data.

Note that VAX RMS also provides each process with local I/O buffers to attain efficient buffering capacity.

3.3.5 Specifying Read-Ahead and Write-Behind

Specifying the read-ahead and write-behind parameters on sequential files can improve performance. These operations require two I/O buffers. (Note that using more than two buffers will usually not improve performance.) These features require the implementation of the *multibuffer* attribute and the specification of two buffers. (See Section 3.3.3).

With most languages, read-ahead and write-behind are the default operations; with others, you will have to specify these operations explicitly by using a clause in the language. If you are using a high-level language that has no clause for specifying read-ahead and write-behind, you must use a VAX MACRO routine when you open the file to set these capabilities.

At the VAX MACRO level, you can select these options by setting the `RAB$V_RAH` bit in the `RAB$L_ROP` field for read-ahead and the `RAB$V_WBH` bit for write-behind prior to requesting the Connect service.

You can use FDL to select these options by using the secondary CONNECT attributes `READ_AHEAD` and `WRITE_BEHIND` respectively.

3.4 Tuning a Relative File

A relative file consists of a file header, a prologue, and a series of fixed-length cells. Each cell contains one record. The first byte in each cell is a deleted record control field; it is followed by the record. Cell size is thus one byte larger than record size.

You can leave cells blank when populating a relative file.

Each cell receives a sequential number. This is the relative record number, which can be used for random access to the record.

Records in a relative file can be fixed, variable, or variable with fixed control (VFC). Fixed-length records are particularly useful in relative files, because the cell size is also fixed.

The maximum size for fixed-length records in a relative file is 16,383 bytes. For variable records the maximum size is 16,381 bytes. The maximum size for VFC records is 16,381 bytes minus the size of the fixed control area. The size of the fixed-control area can be from 1 to 255 bytes.

3.4.1 Bucket Size

With relative files, buckets are the unit of transfer between the disk and memory. You specify bucket size when you create the file but you can subsequently change the size by converting the file (see Chapter 10). You can specify the bucket size using the FDL FILE secondary attribute `BUCKET_SIZE` or by inserting the value directly into the VAX RMS control block fields `FAB$B_BKS` and `XAB$B_BKZ`. Although the size can be as large as 63 blocks, a bucket size larger than one disk track will usually not improve performance.

If you choose to select the bucket size, you should also consider how your application will access the file. For random access, you may want to choose a small bucket size; for sequential access, a large bucket size; and for mixed access, a medium bucket size.

One efficient design for a relative file is to align the file on a cylinder boundary and then specify the size of one disk track as the bucket size. This design requires that you can perform an exact alignment on the file.

If you use the FDL editor to establish the bucket size (this is recommended), the editor will fix the size at the optimal value based on script inputs you provide to it.

The Edit/FDL Utility calculates bucket size by first asking about the file's intended use. If you will be accessing the file randomly, EDIT/FDL sets the bucket size equal to four records because it assumes that four records is a reasonable amount of data for a random access. If you will be accessing records sequentially, EDIT/FDL sets the bucket size equal to 16 records because it assumes that 16 records is a reasonable amount of data for one sequential access.

If you find that your application needs more data per access, then use the EDIT/FDL command MODIFY to change the assigned values

3.4.2 Number of Buffers

The multibuffer count is a run-time parameter which you can set with the DCL command SET RMS_DEFAULT/RELATIVE/BUFFER_COUNT=*n*, the FDL attribute CONNECT MULTIBUFFER_COUNT, or the VAX RMS control block field RAB\$B_MBF. The type of record access to be performed determines the best use of buffers.

The two extremes of record access are when records are processed either completely randomly or completely sequentially. Also, there are cases where records are accessed randomly but may be reaccessed (random with temporal locality) and cases where records are accessed randomly but adjacent records are likely to be accessed (random with spatial locality).

In completely sequential processing, the first record may be located randomly and the following records accessed sequentially (records are usually not referenced more than once). For best performance, you should use one buffer with a large bucket size unless you are using the read-ahead capability which requires two buffers.

Large buckets hold more records, so you can access a greater number of records before doing I/O. However, a small multibuffer count, such as the default of 1 buffer, is sufficient.

Note that you may find tuning the bucket size rather than the number of buffers more helpful when you want to improve performance for sequential access.

Completely random processing means that records are not accessed again, and adjacent records are not likely to be accessed. You should use one buffer with a minimal bucket size. You do not need to build a memory cache because the records are likely to be scattered throughout the file. New requests for records will probably result in an I/O operation, and caching extra buckets only wastes space in your working set.

Performance Considerations

In random with temporal locality processing (reaccessed records), records are processed randomly, but the same records may be accessed again. You should use multiple small buffers to cache records that will be reaccessed. The bucket size can be small for this type of access because the records near the record currently accessed are not likely to be accessed. Caching them in large buckets only wastes space in memory. Multiple buffers allow the previously accessed records to remain in memory for subsequent access.

In random with spatial locality (adjacent records) processing, records are processed randomly, but the next or previous record has a good chance of being accessed. You should use a large buffer and bucket size, which improves the possibility that the next or previous record is in the same bucket as the record just accessed. One or two buffers should be sufficient.

If you process your data file with a combination of these patterns, you should compromise between the processing strategies.

One such application illustrating both temporal and spatial access would be to use the first record in the file as a pointer to the last record accessed. You would read the first record to find the location of the next record to operate on, then go to that record, perform some operation, and update the pointer in the first record. Because you access the first record many times, your access pattern exhibits temporal locality, but because you add records sequentially to the end of the file, your pattern exhibits spatial locality.

When you add records to a relative file, you might consider choosing the deferred write option (FDL attribute `FILE DEFERRED_WRITE`, `FAB$L_FOP` field `FAB$V_DFW`). With this option, the buffer into which the records have been moved is not written to disk until the buffer is needed for other purposes or the file is closed. This option, however, may cause records to be lost if a system crash should occur before the records are written to disk.

To see what the current default buffer count is, give the DCL command `SHOW RMS_DEFAULT`. To set the default buffer count, use the DCL command `SET RMS_DEFAULT/RELATIVE/BUFFER_COUNT=n`, where `n` is the number of buffers.

3.4.3 Global Buffers

If several processes will share the relative file, you may want to specify that the file will use global buffers. A *global buffer* is an I/O buffer that two or more processes can access. If two or more processes are requesting the same information from a file, each process can use the global buffers instead of allocating its own.

Global buffers are not useful in all cases. Forming a memory cache may not reduce the number of I/O operations necessary to process the file. No matter how many global buffers you allocate, VAX RMS always allocates a single I/O buffer per process, which provides efficient buffering capacity.

If your application has several processes sharing the file and accessing the same records in a transaction sequence, then you may benefit from allocating enough global buffers to cache these shared records.

3.4.4 Using Deferred Write

Deferred write is a run-time option that can improve performance. It is the default operation with some languages and can be specified by clauses in other languages. If there is no language support, you can call a VAX MACRO subroutine that sets the FAB\$L_FOP field, the FAB\$V_DFW option before opening the file.

In a deferred write, VAX RMS delays the writing of a modified bucket to disk until the buffer is needed for another purpose or until another process needs to use the bucket. This deferment improves performance if a subsequent record access refers to the same bucket because it reduces the number of disk I/O operations required. The largest performance gains come from using deferred write with sequential access.

For example, in a relative file with 100-byte records and 2-block buckets, 10 records will fit in one bucket. Without deferred write, writing records 1 through 10 in order would result in eleven I/O operations — one for the initial file access and one each for the records.

With deferred write, you need only two I/O operations — one for the initial file access and one to write the bucket.

A larger cache might be useful in situations where the accesses are not strictly sequential, but follow some local pattern.

3.5 Tuning an Indexed File

This section discusses the structure of indexed files and ways to optimize their performance.

3.5.1 File Structure

An indexed file consists of a file header, a prologue, and one or more index structures. The primary index structure contains the data records. If the file has alternate keys, then it will have an alternate index structure for each alternate key. The alternate index structures contain secondary index data records (SIDRs) that provide pointers to the data records in the primary index structure. The index structures also contain the values of the keys by which VAX RMS accesses the records in the indexed file.

3.5.1.1 Prologues

VAX RMS places information concerning file attributes, key descriptors, and area descriptors in the prologue. You can examine the prologue with the Analyze/RMS_File Utility described in Chapter 10.

There are three types of prologues — Prolog 1, Prolog 2, and Prolog 3.

Prolog 1 and 2

Any indexed file created with a version of VAX/VMS earlier than Version 3.0 will be either a Prolog 1 or a Prolog 2 file. Prolog 1 and Prolog 2 files operate identically; there is no difference between them that the user can detect.

If your indexed file has all keys of the string data type, then the file will be a Prolog 1 file. If the indexed file has any key of a numeric type, it will be a Prolog 2 file. You cannot specify 8-byte numeric keys for a Prolog 2 file.

You cannot use the Convert/Reclaim Utility on a Prolog 1 or Prolog 2 file to reclaim empty buckets. If your file undergoes a large number of deletions, resulting in empty, unusable buckets, you must use the Convert Utility (CONVERT) to reorganize the file. (Note that CONVERT will establish new RFAs for the records.)

The compression allowed with Prolog 3 files is not possible with Prolog 1 or Prolog 2 files.

Prolog 3

Prolog 3 files can accept multiple (or alternate) keys and all data types (including the nonstring 8-byte BIN8 and INT8 types). They also give you the option of saving space by compressing your data, indexes, and keys.

Key compression compresses the key values stored in the data buckets. Likewise, index compression compresses the key values stored in index buckets. Data compression, on the other hand, compresses the data portion of the records in the data buckets.

When keys are compressed in index or data records, repeating leading and trailing characters are compressed. With front key compression, any characters which are identical to the characters at the front of the previous key are compressed. For example, the keys JOHN, JOHNS, JOHNSON, and JONES would appear as JOHN, S, ON, and NES.

With rear key compression, any repeating characters at the end of the key are compressed to a single character. For instance, The key JOHNSON00000 would appear as JOHNSON0.

With data compression, instances of five or more repeated characters in a row are compressed to a single character.

Compression has a direct effect on CPU time and disk space. Compression increases CPU time, but the keys are smaller, so your application can scan more quickly through the data and index buckets.

The disk space saved by using Prolog 3 indexed files can significantly improve performance. With compression, each I/O buffer can hold more information and so improve buffer space efficiency. Compression can also decrease the number of index levels, which also decreases the number of I/O operations per random access.

Prolog 3 files can have segmented primary keys, but the segments cannot overlap. If you want to use a Prolog 3 file in this case, consider defining the overlapping segmented key as an alternate key and then choosing a different key to be the primary key. If you want to use overlapping primary key segments, you must use a Prolog 2 file.

If record deletions result in empty buckets in Prolog 3 files, you can use the Convert/Reclaim Utility to make the buckets usable again. Because CONVERT/RECLAIM does not create a new file, RFAs remain the same.

Note that RMS-11 does not support Prolog 3 files. To use a Prolog 3 file with RMS-11 you must first use the Convert Utility to make the file a Prolog 1 or Prolog 2 file.

3.5.1.2 Primary Index Structure

The primary index structure consists of the file's data records and a key pathway based on the primary key (key 0). The base of a primary index structure is the data records themselves, arranged sequentially according to the primary key value. The data records are called level 0 of the index structure.

The data records are grouped into buckets, which is the I/O unit for indexed files. Because the records are arranged according to their primary key values, no other record in the bucket has a higher key value than the last record in that bucket. This high key value, along with a pointer to the data bucket, is copied to an *index record* on the next level of the index structure, known as level 1.

The index records are also placed in buckets. The last index record in a bucket itself has the high key value for its bucket; this high key value is then copied to an index record on the next higher level. This process continues until all of the index records on a level fit into one bucket. This level is then known as the root level for that index structure.

Figure 3-1 is a diagram of an index structure.

Figure 3-2 illustrates a primary index structure. (For simplicity's sake, the records are assumed to be uncompressed, and the contents of the data records are not shown.) The records are 132 bytes long, with a primary key field of 6 bytes. Bucket size is one block, which means that each bucket on Level 0 can contain 3 records. The calculation of the number of records per bucket in this case is shown below.

$$\begin{aligned} 512 \text{ bytes} - 15 \text{ bytes of overhead} &= 497 \text{ bytes} \\ 497 / 132 &= 3.77 \end{aligned}$$

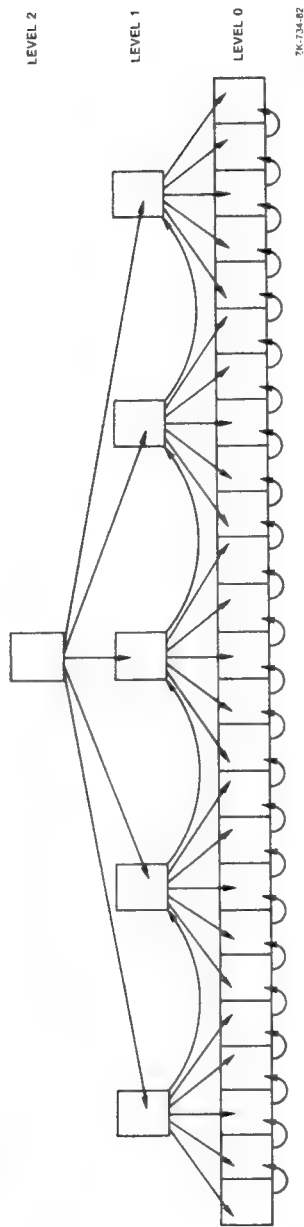
You must round the remainder to the lowest integer, which is 3.

Because the key size is small and the database in this example consists of only 27 records, the index records can all fit in one bucket on level 1. The index records in this example are 6 bytes long. Each index record has one byte of control information. In this example, the size of the pointers is 2 bytes per index record, for a total index record size of 9 bytes. The calculation of the number of records per bucket is shown below.

$$\begin{aligned} 512 \text{ bytes} - 15 \text{ bytes of overhead} &= 497 \text{ bytes} \\ 497 / 9 &= 55.2 \end{aligned}$$

Here again, you must round the remainder to the lowest integer, 55.

Figure 3-1 VAX RMS Index Structure



If you wanted to read the record with the primary key 14, VAX RMS would begin by scanning the root level bucket, looking for the first index record with a key value greater than or equal to 14. This record would be the index record with key 15. The index record contains a pointer to the level 0 data bucket that contains the records with the keys 13, 14, and 15. Scanning that bucket, VAX RMS would find the record (see Figure 3-3).

3.5.1.3 Alternate Index Structures

The alternate index structures are similar to the primary index structure. Instead of containing data records, though, they contain secondary index data records (SIDRs). A SIDR includes an alternate key value from a data record stored in the primary index and one or more pointers to data records in the primary index. (The SIDRs have pointers to more than one record only if you have allowed duplicate keys and if there actually are records with the same key value in the database.)

You do not need a SIDR for every data record in the database. If a variable-length record is not long enough to contain a given alternate key, a SIDR will not be created. For example, if you define an alternate key field to be bytes 10 through 20 and you insert a 15-byte record, no SIDR is created in that alternate index structure.

You can define a null value when you create the file. For the string data type, you can specify any character although the ASCII null character is the default. For numeric keys, the null value is zero (0). When you define a null value for an alternate key, records containing the null value for every character in the key field will not be included in the alternate index.

A file with alternate index structures requires more disk space than a file without.

3.5.1.4 Records

Records in an indexed file can be fixed length or variable length. Fixed-length records begin with a record header. Variable-length records include a record header followed by 2 bytes of record length overhead. Unlike relative files, each variable-length record in an indexed file requires only enough space for the record. See Table 2-2 for more information on record overhead.

Records cannot span bucket boundaries.

Figure 3-2 A Primary Index Structure

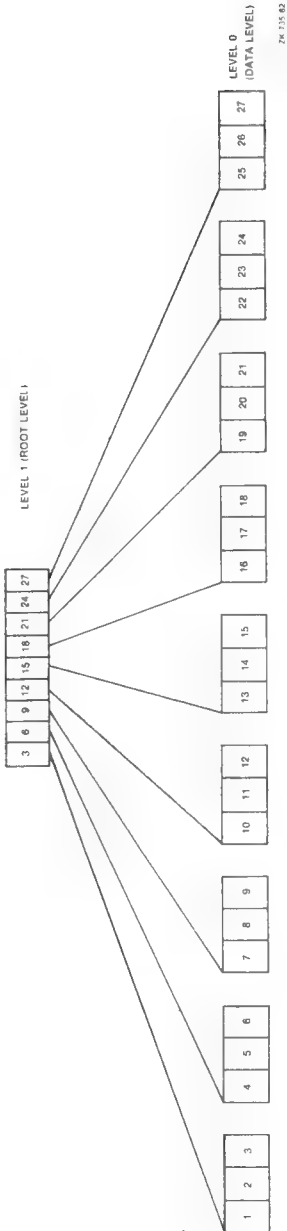
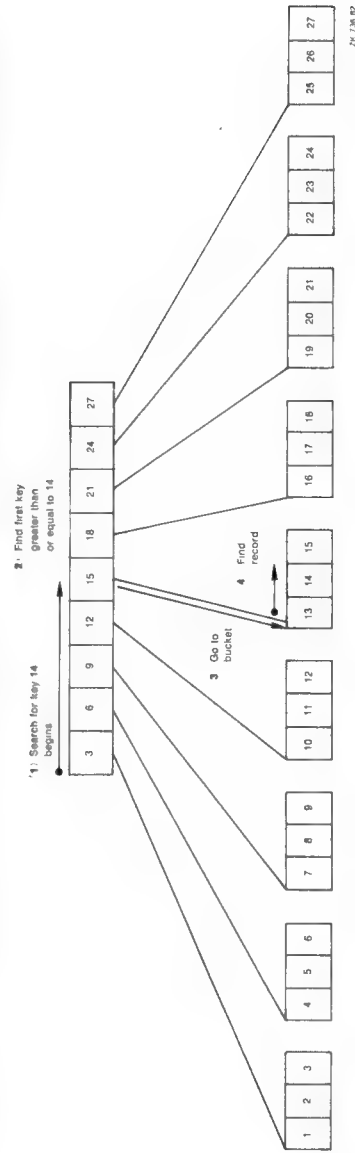


Figure 3-3 Finding the Record with Key 14



For Prolog 3 files, the maximum record size is 16,300 bytes. For Prolog 1 and Prolog 2 files, the largest size for a fixed-length record is 16,362 bytes; the largest size for a variable-length record is 16,360 bytes. Note that when you specify a record size for a PROLOG 3 file that is greater than the maximum record size, VAX RMS automatically converts the file to a PROLOG 1 or PROLOG 2 file.

Your record size should reflect application requirements. There is no advantage to using a record size that is based on the number of bytes in a bucket.

The data in the records must contain the value of the primary key. The records can contain either a valid key field value for the alternate keys or, if you specify that null keys are allowed, a field of null characters.

3.5.1.5 Keys

A key is a record field that identifies the record to help you retrieve the record. There are two types of keys — primary and alternate. Data records are stored in the file in the order of their primary key. The most time-efficient value for primary keys is a unique value that begins at byte 0 of the record. You can allow duplicate keys in the primary index but duplicates may slow performance.

Both the primary and alternate keys can be character strings or numerical values. Key type is specified by the FDL attribute KEY TYPE.

If it is not possible to put the records into the file in order of their primary key, you should specify that the buckets not be filled completely when the file is loaded. If you attempt to write a record to a full bucket, a bucket split occurs. VAX RMS keeps half of the records in the original bucket and moves the other records to the newly created bucket. Each record that moves leaves behind a pointer to the new bucket called a record reference vector (RRV). You should avoid bucket splits because they use additional disk space and CPU time. An extra I/O operation is required to access a record in a split bucket when the program accesses a record by an alternate key or by RFA.

Alternate keys have a direct impact on I/O operations, CPU time, and disk space. The number of I/O operations and the CPU time required for Put, Update, and Delete operations is directly proportional to the number of keys. For example, inserting a record with a primary key and three alternate keys takes approximately four times longer than inserting a record with only a primary key.

To update the value of an alternate key, you have to traverse the alternate index structure twice, and bucket splits are more likely to occur. Accessing an alternate key randomly generally requires an extra I/O operation over a comparable access by the primary key, and extra disk space is required to store each alternate index structure.

Alternate keys are more likely than primary keys to have duplicate values. For example, the zip code is a common alternate key. However, allowing many duplicate values can have a performance cost. They can cause clustered record or pointer insertions in data buckets, long sequential searches, a high number of I/O operations, and loss of physical contiguity due to continuation buckets (especially for the primary key).

Where possible, you should validate record keys before inserting the record, especially when you have primary and alternate keys. Note too that you can avoid having to update your index structure by preventing errors,

In general, as the number of keys increases, so does the time it takes to add and delete records from your file. If CPU time is a critical resource on your system, you should define as few keys as possible.

If you are reading records in your file, the number of keys has relatively little impact on performance.

3.5.1.6 Areas

An *area* is a portion of an indexed file that VAX RMS treats as a separate entity. Using multiple areas has distinct advantages. You can divide an indexed file into separate areas where each area has its own bucket size, initial allocation, extension size, and volume positioning, just as if each area were a separate file. However, if each area has a different bucket size, all buffers will be as large as the largest bucket. If you use multiple areas, the file itself will probably not be contiguous; however, you can make each area within the file contiguous by specifying the FDL attribute AREA CONTIGUOUS. To ensure that the area is created without error, use the AREA BEST_TRY_CONTIGUOUS attribute.

When you separate key and data areas, you tend to keep related buckets close together thereby decreasing disk seek time. You also minimize the number of disk-head movements for a series of operations. For example, if you have a dedicated multidisk volume set, you could place the data level of a file in an area on one disk and the index levels of the file in an area on a separate disk. Then there would be little or no competition for the disk head on the disk that contains the index structures.

One recommended strategy is to allocate a separate area for level 0 of a primary index (the data level). These buckets are the only ones referenced when you access the records sequentially by their primary key, so keeping them in a separate area optimizes that type of operation.

Do not allocate separate areas for level 1 of an index and the other index levels if the index has just one level. In such a case, you would force VAX RMS to create an additional level in the index structure.

In most cases, you should allocate at least one area for each alternate index structure. By default, EDIT/FDL will create two areas for each index structure in an indexed file — one for the data level and one for all of the index levels. You can allocate up to 255 areas, so with most applications you can set up enough areas to handle all alternate index structures.

It is possible to set up a separate area for each of the following:

- Primary index level 0 (the data records)
- Primary index level 1 (the lowest index level)
- Primary index levels 2+ (the rest of the index levels)
- Alternate index level 0 (the secondary index data records)
- Alternate index level 1 (the lowest index level)
- Alternate index levels 2+ (the rest of the index levels)

Be sure to allocate sufficient space for each area and to specify area contiguity, because extending an area generally creates a noncontiguous area extent. The resulting noncontiguous extent may be anywhere on the disk, and you may lose the benefits of multiple areas.

If you are using a single area for the file, you should allocate enough contiguous space for the entire file at creation time. If you plan to add data to the file later on, you should allocate extra space. The FDL attribute FILE ALLOCATION sets this value. To make sure the allocation is contiguous, set the FDL attribute FILE CONTIGUOUS to YES.

If you are using multiple areas, you should allocate each one by specifying a value for the FDL attribute AREA ALLOCATION.

If the file is relatively small or if you know that it will need to be extended, you do not have to use multiple areas. In such cases, it is more important to calculate the proper extension size.

To specify multiple areas using an FDL file, you assign each area its own AREA primary attribute. The AREA primary attribute takes as an argument a number whose value identifies the area.

Use the KEY primary attribute with its secondary attributes DATA__AREA, LEVEL1__INDEX__AREA, and INDEX__AREA to match each area specified with its index level. Key 0 is known as the primary key. Therefore, in the primary index structure, the primary attribute KEY must take the value 0. Then, within the KEY 0 section, you assign to the DATA__AREA secondary attribute the number that you used to define the area where you want the data records to appear.

You then match the KEY LEVEL1__INDEX__AREA secondary attribute with an AREA primary attribute by assigning the appropriate area number to the LEVEL1__INDEX__AREA secondary attribute. You also assign the number of an area to the INDEX__AREA secondary attribute for the other index levels in the primary index structure. For each alternate index structure, you use the same secondary attributes (DATA__AREA, LEVEL1__INDEX__AREA, INDEX__AREA) in another KEY primary attribute. In KEY sections that define alternate keys, the DATA__AREA is where VAX RMS will put the SIDs.

3.5.2 Optimizing File Performance

This section discusses adjustments in the file design that can improve the file's performance.

3.5.2.1 Bucket Size

For indexed files, the bucket size controls the number of levels in the index structure, which has the greatest impact on performance for most applications. You can specify the bucket size with the FDL attribute FILE BUCKET_SIZE or the VAX RMS control block fields FAB\$B_BKS and XAB\$B_BKZ.

The number of index levels should rarely exceed four, so set your bucket size accordingly. In general, the smaller the bucket size, the deeper the tree structure; if you find that a small increase in bucket size will eliminate one level, then use the larger bucket size. At some point, however, the benefit of having fewer levels will be offset by the cost of scanning through the larger buckets.

As a rule, you should never increase bucket size unless the increase reduces the number of levels. For example, you may find that a bucket size of 4 or more yields an index with four levels, and a bucket size of 10 or more yields an index with three levels. In this case, you would never want to specify a bucket size of 9 because that would not reduce the number of levels, and performance would not improve. In fact, such a specification could hurt

performance because each I/O operation would take longer, yet the number of accesses would remain the same. However, larger bucket sizes will always improve performance if you are accessing the records sequentially by primary key because more records will then fit in a bucket.

On the other hand, with smaller buckets, you have to search fewer keys. So if you can cache your whole structure (except for level 0), you can save a lot of time. Also, performance in this case is comparable to flat file design although add operations may take a little longer.

You can decrease the depth of your index structure in two ways. First, you can increase the number of records per bucket by increasing the bucket size, increasing the fill factor, using compression, or decreasing the size of keys and records.

However, these methods also have disadvantages. Larger buckets use more buffer space in memory. And the number of records per bucket determines bucket search time, which directly affects CPU time. A larger fill factor decreases the room for growth in the file, so bucket splits may occur. Compression increases the record search time.

Alternatively, you can reduce the index depth by decreasing the number of records in the file.

If you are using multiple areas, you can set a different bucket size for each area. You should use different bucket sizes if you are performing random accesses of records in no predictable pattern and the data records are large. Using different bucket sizes allows you to specify a smaller size for the index structures and SIDRs than for the primary data level.

You can use the Edit/FDL Utility to determine the optimum bucket size.

Use the same bucket size for all areas if the data records are small or if the record accesses follow a clustered pattern, that is, if the records that you access have keys that are close in value.

In general, decreasing the bucket size increases other resources:

- Levels in the tree structure
- Buckets needed to maintain the tree structure
- Buffers needed for cache

On the other hand, decreasing the bucket size *decreases* the pages per bucket and the average number of keys searched while traversing the tree.

3.5.2.2 Fill Factor

If you know that the application will require random insertions into the database, you should reserve some space in the buckets when records are first loaded into the file. You can specify a fill factor from 50% to 100%. For example, a fill factor of 50% means that VAX RMS will write records in only half of each bucket when the records are first loaded, leaving the remainder of the bucket empty for future write operations. This fill factor will minimize the number of bucket splits that will occur.

Fill factor is set with the FDL attributes KEY DATA_FILL and KEY INDEX_FILL. The value assigned to both attributes should be the same.

When you specify a fill factor, consider the following:

- If the inserted records are distributed unevenly (highly skewed) by their primary key value, then specifying a fill factor of less than 100% will not reduce the number of bucket splits.
- If the records have key values that are close or if they are added at one end of the file, many bucket splits will occur anyway, and the partially filled buckets in the database will just be wasted space. If this is the case, you should either specify a fill factor of 100% and then use the Convert Utility to reorganize the file after the insertions are made, or you should choose a different primary key.
- If the inserted records are distributed fairly evenly or by their primary key, then specifying a fill factor of less than 100% could significantly reduce bucket splits. However, the trade-off is initially wasted disk space.

3.5.2.3 Number of Buffers

At run time, you can specify the number of buffers with the FDL attribute CONNECT MULTIBUFFER_COUNT or the VAX RMS control block field RAB\$B_MBF. The number of buffers each application needs depends on the type of record access your application performs.

Two buffers is the minimum value for indexed files. If your application performs sequential access on your database, two buffers are sufficient. More than two buffers for sequential access could actually degrade performance. During a sequential access, a given bucket will be accessed as many times in a row as there are records in the bucket. After VAX RMS has read the records in that bucket, the bucket will not be referenced again. Therefore, it is unnecessary to cache extra buckets when accessing records sequentially.

When you access indexed files randomly, VAX RMS must read the index portion of the file to locate the record you want to process. VAX RMS tries to keep the higher level buckets of the index in memory; the buffers for the actual data buckets and the lower level index buckets tend to be reused first when other buckets need to be cached. Therefore, you should use as many buffers as your process working set can support so you can cache as many buckets as possible.

When you access records sequentially, even after you have located the first record randomly, you should use a large bucket size. A small multibuffer count, such as the default of 2 buffers, is sufficient.

If you process your data file with a combination of the above access modes, you should compromise on the recommended bucket sizes and number of buffers.

When you add records to an indexed file, consider choosing the deferred write option (FDL attribute `FILE DEFERRED_WRITE`; `FAB$L_FOP` field `FAB$V_DFW`). With this option, the buffer into which the records have been moved is not written to disk until the buffer is needed for other purposes or the file is closed. This option, however, may cause records to be lost if a system crash should occur before the records are written to disk.

To see what the current default buffer count is, give the DCL command `SHOW RMS_DEFAULT`. To set the default buffer count, use the DCL command `SET RMS_DEFAULT/INDEXED/BUFFER_COUNT=n`, where `n` is the number of buffers.

In general, you must consider several trade-offs when you set the number of buffers your application needs:

- CPU time
- Availability of memory and number of page faults
- I/O operations

With indexed files, buckets (not blocks) are the unit of transfer between the disk and memory. You specify the bucket size when you create the file although you can change the bucket size of an existing file with the Convert Utility (see Chapter 10).

3.5.2.4 Global Buffers

If several processes will share the indexed file concurrently, you may want to specify that the file will use global buffers. A *global buffer* is an I/O buffer that two or more processes can access. If two or more processes are requesting the same information from a file, each process can use the global buffers instead of allocating its own.

Only one copy of the buffers is resident at any time in memory although the buffers are charged against each process's working set size.

The general guideline for using global buffers is the same as for the local process I/O buffers. However, global buffers only provide significant benefits if more than one process references a particular bucket in the global cache without performing an I/O operation to read it into memory. Therefore, if multiple processes are independently reading records and are using sequential access, they will most likely be referencing separate buckets. In that case, the total number of I/O operations will not be reduced, so global buffers will not improve performance.

The use of global buffers with deferred write processing has some implications, which are described in Section 3.5.2.5.

3.5.2.5 Using Deferred Write

Deferred write is a run-time option that can improve performance. It is the default operation with some languages and can be specified by clauses in other languages. If there is no language support, you can call a VAX MACRO subroutine that sets the FAB\$L_FOP field, the FAB\$V_DFW option.

In a deferred write, VAX RMS delays the writing to disk of a modified bucket until the buffer is needed to read another bucket into the cache, or until another process needs to reference the modified bucket. If a subsequent operation references the bucket before it is flushed out to disk, then one I/O operation has been eliminated. Typically, the largest performance gains come from using deferred write with sequential access, because random accesses of the file usually result in several I/O operations to bring in the single records.

Global buffers are not used directly to retain modified information when deferred write is enabled. If a global buffer is modified and deferred write is enabled, the contents of the global buffer will be copied to a process local buffer before other processes are allowed to access the global buffer contents. Subsequent use of the modified buffer by the process that deferred the writeback of it will reference the process local buffer while it remains in the process local cache. Reference to the global buffer by another process will cause the contents of the process local buffer to be written back to disk.

If a global buffer is modified and deferred write is not enabled, then the contents will simply be written out to disk from the global buffer. Therefore, using global buffers along with deferred write may cause a slight increase in processing overhead if in fact no further references to the modified buffer occur before it is flushed from the cache anyway. For that reason, you may want to disable deferred write for processes that do not reaccess buffers after records have been written to them.

Not all operations on indexed files can be deferred. Any operation that causes a bucket split will force the writeback of the modified buckets to disk. (This forced writeback decreases the chances of lost information should a system failure occur.)

Using deferred write will usually improve performance. Consider the following example. The indexed file has a single key and its records are 100 bytes long. The bucket size is 3 blocks with a fill factor of 67%. Thus, there is an average of 10 records in each bucket. A batch program reads each record and updates part of it, beginning at the first record in the file and moving through the records sequentially. Without deferred write, 11 disk I/O operations would occur for every 10 records — one to read the bucket and one to write the bucket for each record. With deferred write, only two disk I/O operations would occur for every 10 records — one to read the bucket and one to write the bucket after the record operations had been completed.

3.6 Processing in a VAXcluster

This section discusses aspects of designing file applications for a multiple node VAXcluster and the performance you can reasonably expect from this environment.

Processing in a VAXcluster environment offers many advantages:

- **Performance** — In general, the performance of each node in a VAXcluster is similar to that of a single-node system that has the same processing load assuming the aggregate I/O per disk drive is reasonable.
- **Availability** — With the appropriate configuration, a node that leaves the cluster does not stop the cluster from running.
- **Flexibility** — You can process shared applications on more than one node.
- **Accessibility** — Shared resources are very easy to use in a cluster. The synchronized access to the data provides data integrity with no redundancy.

For more information on VAXclusters in general, see the *Guide to VAXclusters*.

AA-Y508B-TE

3.6.1 VAXCluster Shared Access

Shared access is one of the chief advantages of processing in a cluster environment. Many applications that run on a single-node system can run on a multiple-node cluster with no changes.

However, applications which access shared files in a cluster incur some additional overhead for the cluster synchronization; the amount is dependent upon the locking requirements of your application.

3.6.1.1 Locking Considerations

The distributed lock manager allows you to share files concurrently in an organized manner with other users. VAX RMS uses the lock manager to control file access.

The *lock-mastering node* controls the record and bucket locking for a given file for users on every node of the cluster. Initially, it is the first node from which the file is opened. However, another node may become the lock-mastering node when a node either joins or leaves the cluster.

The lock-mastering node may also change every time the file is opened. When another process opens the file (provided that the file was closed), the node on which that process resides becomes the new lock-mastering node for that file.

Lock requests issued by processes on the lock-mastering node incur less cost than lock requests issued from other nodes. On the other hand, the lock-mastering node has the additional work of processing lock requests for that file for all other nodes.

The *lock-requesting node* is any node in the cluster *other* than the lock-mastering node for a given file.

VAX RMS locks buckets and records during record operations only if the file is open for shared writing. On the other hand, VAX RMS does no locking during record operations if the file is open for shared read-only access or for exclusive access.

Lock requests for *root locks* (top-level or parent locks) in a cluster may be slightly slower than on a single-node system. However, these locks are used when you open and close files, so the time for lock operations is only a fraction of the total time needed to open and close files.

There is no performance difference between a single-node system and a VAXcluster if the file sharing takes place on a single node of the cluster. Only when sharing spans across the cluster nodes does distributed locking occur.

Performance Considerations

As a result, the record locking itself may take a little longer, but since you have multiple CPUs in the cluster, your application gets the benefit of the added processing power.

Sharing files in a cluster also requires enough memory for nonpaged pool to store additional lock data structures. This requirement, however, is dependent upon your processing load.

3.6.1.2 I/O Considerations

Sharing in a cluster environment also means sharing resources such as disks and other pieces of I/O hardware. With applications on many nodes sharing the data on one disk, the applications can easily become I/O bound. As a result, if many users from many nodes are trying to access the shared disk at the same time, the disk can become a bottleneck limiting the performance of the cluster. In other words, the performance is dependent upon your workload.

3.6.2 Performance Recommendations

Four general recommendations about performance in a VAXcluster environment are discussed below.

Estimate how much I/O your application needs to do. In a cluster, and particularly with a shared file, multiple nodes can generate a lot of I/O requests to a single disk. The capacity of the disk to handle I/O traffic can affect cluster performance if you allow your applications to become I/O bound. The Monitor Utility is a good tool for estimating how many I/O requests your application generates. For more information on the Monitor Utility, see the *VAX/VMS Monitor Utility Reference Manual*.

Processing a file with exclusive access gives you better performance than with shared-write access. Opening files for unnecessary shared-write access incurs needless locking cost (even on a single node system).

If your application can be performed on a single CPU and if sufficient CPU resources and I/O capacity are available, your application will perform faster than if it were spread over many nodes.

Space overhead for the lock database and other system software can be significant, so make sure you have sufficient memory.

4

Creating and Populating Files

After you have designed your file, you need to create it. First you must specify the file characteristics you selected during the design phase. Then you need to create the actual file with those characteristics and to protect it (decide who has access to the file). Last, you need to put records in the file, or "populate" it.

This chapter describes the process of creating and populating files. Section 4.1 tells how to select and specify file-creation characteristics. Section 4.2 describes how to create a file. Section 4.3 explains how to define file protection, and Section 4.4 describes how to populate the file. A summary of the options related to file creation is provided in Section 4.5.

4.1 File Creation Characteristics

There are two ways you can specify the characteristics you need to create a file. If you are using VAX MACRO or BLISS-32, you can specify file creation characteristics by including VAX RMS control blocks in your application program.

If you are using a high-level language, you can use the File Definition Language (FDL), which is a special-purpose language that is used to write specifications for data files. Of course, you also have the option of using FDL with MACRO or BLISS-32.

The following sections describe how you can specify file-creation characteristics by using VAX RMS control blocks or by creating FDL files.

4.1.1 Using VAX RMS Control Blocks

You can establish characteristics for the file you create by using a VAX RMS file access control block (FAB) and VAX RMS extended attribute control blocks (XAB). These control blocks allow you to take the defaults that VAX RMS provides, or to override the defaults and define the characteristics that suit your particular application.

4.1.1.1 File Access Block

The file access block (FAB) is made up of fields that describe various file characteristics and contain file-related information including:

- The addresses of file name and default name strings
- The file organization
- The record format
- Information about disk storage space

The FAB lets you use both the creation-time and run-time characteristics of VAX RMS. You must define one FAB for each file your program will open or create.

For more information on the file access block, see the *VAX Record Management Services Reference Manual*.

4.1.1.2 Extended Attribute Blocks

Extended attribute blocks (XABs) are optional user control blocks that contain supplementary file-attribute information. The various XABs can be used to declare the following information:

- Initial size and extent information (XABALL)
- File protection (XABPRO)
- Key definition (XABKEY)
- Date and time information (XABDAT)

Like the FAB, the XABs allow you to use both the creation-time and run-time characteristics of VAX RMS.

With the XABs, you can define various file attributes beyond those specified in the FAB.

For more information on the extended attribute blocks, see the *VAX Record Management Services Reference Manual*.

4.1.2 Using File Definition Language

FDL gives you a very convenient way to create data files through the use of special text files, generally called FDL files. FDL files specify appropriate of file attributes and values written in File Definition Language. You create and modify these FDL files using a special part of the FDL Facility called the Edit/FDL Utility.

Subsequently, you can then use the Create/FDL and Convert utilities to create data files from the specifications in the FDL files. In addition, you can use the Analyze/RMS_File Utility to create FDL files from existing data files. FDL files created in this manner contain special analysis sections that you can use with EDIT/FDL to tune your data files.

By using an FDL file to create a data file from a higher-level language, you have access to most of the VAX RMS creation-time characteristics without requiring direct access to the VAX RMS control blocks (FABs and XABs). However, to use the full set of VAX RMS connect-time capabilities including file specification wild card characters, you must use the VAX RMS control blocks.

The Edit/FDL Utility contains built-in design algorithms to help you optimize data file design. EDIT/FDL recognizes correct FDL syntax and informs you immediately of syntax errors. (You can use a text editor or the DCL command CREATE to create an FDL file, but you must then follow the validity rules listed in the *VAX/VMS File Definition Language Facility Reference Manual*.)

4.1.2.1 Using the Edit/FDL Utility

There are two ways you can use the Edit/FDL Utility — with a terminal dialog (interactively) or without one (noninteractively).

If you use EDIT/FDL noninteractively, you can execute only the OPTIMIZE script. This option allows you to optimize an existing FDL file without going through an interactive session. For more information, see Section 10.3.

On the other hand, if you use EDIT/FDL interactively, you can use all the scripts, each of which has a series of menus. When you first enter the utility, you get a main menu that displays the list of available Edit/FDL commands. When you make a menu selection, you need only enter the first letter because the first character of each selection is unique.

Table 4-1 summarizes the Edit/FDL utility commands.

Table 4-1 Summary of Edit/FDL Utility Commands

Command	Function
ADD	Inserts one or more lines into the FDL definition. If the line already exists, you can replace it with your new line. Once you have inserted a line, you can continue to add lines until you are satisfied with that particular primary section. If no primary section exists to hold the secondary attribute being added, then EDIT/FDL creates one.
DELETE	Removes one or more lines from the FDL definition. If you delete all of the secondary attributes in a primary section, you effectively remove the primary attribute. Once you have removed a line, you can continue to delete lines under that particular primary section.
EXIT	Creates the output FDL file, stores the current FDL definition in it, and terminates the EDIT/FDL session. EDIT/FDL leaves unchanged any FDL file that it used as input. The FDL file that is created is, by default, a sequential file with variable-length records and carriage-return record attributes, and has your process's default VAX RMS protection and ownership. To change these default settings, see Section 4.1.2.3.
HELP	Displays the top level help text for EDIT/FDL and then continues to prompt for more keywords. Pressing the RETURN key in response to the "Topic?" prompt or pressing CTRL/Z will return you to the main function prompt.
INVOKE	Prompts you for your choice of scripts and starts a series of logically ordered questions that help you create new FDL files or modify existing ones.
MODIFY	Allows you to change the value of one or more lines in the FDL definition. Once you have changed a line, you can continue to modify lines under that particular primary section.
QUIT	Aborts the session without creating an output FDL file. You can also press CTRL/C or CTRL/Y to abort the session.
SET	Allows you to establish defaults or to select any of the FDL editor characteristics you forgot to specify on the command line.

Table 4-1 (Cont.) Summary of Edit/FDL Utility Commands

Command	Function
VIEW	Displays the current FDL definition, which is what would be put into the output FDL file if you gave the EXIT command.
?	Causes the utility to display more information about that question. You can enter the question mark character in response to any question asked by EDIT/FDL. In all cases, it will result in repetition of the question. When you make an invalid response to an EDIT/FDL question, the utility's response will be the same as if you had entered a question mark.

CTRL/Z has the same effect as the EXIT command if you use it at the main menu level. If you use it from any other level, CTRL/Z returns you to the main editor function prompt.

In most cases, entering a command causes EDIT/FDL to generate another menu. For instance, typing the ADD command causes the menu below to be displayed.

Legal Primary Attributes

ACCESS attributes set the run-time access mode of the file
 AREA x attributes define the characteristics of file area x
 CONNECT attributes set various VAX RMS run-time options
 DATE attributes set the date parameters of the file
 FILE attributes affect the entire VAX RMS data file
 KEY y attributes define the characteristics of key y
 RECORD attributes set the non-key aspects of each record
 SHARING attributes set the run-time sharing mode of the file
 SYSTEM attributes document operating system-specific items
 TITLE is the header line for the FDL file

Enter desired primary (Keyword) [FILE] :

However, one of the most important features of EDIT/FDL is that it helps you create FDL files that define indexed, relative, and sequential data files. To this end, EDIT/FDL has seven scripts that contain questions to guide you through the interactive session. You can choose one of these scripts at the start of a session, or you can instruct EDIT/FDL to automatically invoke one of the scripts every time that you give the EDIT/FDL command.

Table 4-2 lists the seven scripts.

Table 4-2 EDIT/FDL scripts

Script	Function
ADD_KEY	Allows you to model or add to the attributes of a new index.
DELETE_KEY	Allows you to remove attributes from the highest index of your file.
INDEXED	Begins a dialog in which you are prompted for information about the indexed data file you want to create from the FDL file. EDIT/FDL will supply values for certain attributes.
OPTIMIZE	Helps you redesign an FDL file that was created with the Analyze/RMS_File Utility. The FDL file itself is one of the inputs to the Edit/FDL utility. In effect, this script allows you to tune the parameters of your indexes using the file statistics from the FDL ANALYSIS sections produced by ANALYZE /RMS_FILE.
RELATIVE	Begins a dialog in which you are prompted for information about the relative data file to be created from the FDL file. EDIT/FDL will supply values for certain attributes.
SEQUENTIAL	Begins a dialog in which you are prompted for information about the sequential data file to be created from the FDL file. EDIT/FDL will supply values for certain attributes.
TOUCHUP	Begins a dialog in which you are prompted for information about the changes you wish to make to an existing index.

An interactive session is controlled by these EDIT/FDL scripts. You can invoke a script in two ways:

- You can give the INVOKE command from the main menu and then choose your script. After you supply values in response to the script questions, EDIT/FDL displays a list of FDL attributes and their assigned values. At this point, you can use EDIT/FDL commands to further modify the attribute values or to end the editing session.
- You can automatically begin a script by entering the DCL command
EDIT/FDL/SCRIPT=script-name

This command bypasses the main menu and directly displays the menu for the script you have named.

Example 4-1 below shows a sample session with the FDL Editor.

Example 4-1 Sample FDL Edit Session

VAX-11 FDL Editor

```

Add      to insert one or more lines into the FDL definition
Delete   to delete one or more lines from the FDL definition
Exit     to leave the FDL Editor after creating the FDL file
Help     to obtain information about the FDL Editor
① Invoke to initiate a script of related questions
Modify   to change existing line(s) in the FDL definition
Quit     to abort the FDL Editor with no FDL file creation
Set      to specify FDL Editor characteristics
View     to display the current FDL Definition

② Main Editor Function      (Keyword)[Help] : INVOKE

      Script Title Selection
Add_Key      modeling and addition of a new index's parameters
Delete_Key   removal of the highest index's parameters
Indexed      modeling of parameters for an entire Indexed file
③ Optimize   tuning of all indexes' parameters using file statistics
Relative     selection of parameters for a Relative file
Sequential   selection of parameters for a Sequential file
Touchup      remodeling of parameters for a particular index

④ Editing Script Title      (Keyword)[-] : INDEXED
⑤ Target disk volume Cluster Size (1-1Giga)[3] : 3
⑥ Number of Keys to Define   (1-255)[1] : 1

Line  Bucket Size vs Index Depth      as a 2 dimensional plot
Fill  Bucket Size vs      Load Fill Percent      vs Index Depth
⑦ Key  Bucket Size vs      Key Length      vs Index Depth
Record Bucket Size vs      Record Size      vs Index Depth
Init  Bucket Size vs Initial Load Record Count vs Index Depth
Add   Bucket Size vs Additional Record Count vs Index Depth

⑧ Graph type to display      (Keyword)[Line] : LINE

```

(Continued on next page)

Example 4-1 (Cont.) Sample FDL Edit Session

```

①Number of Records that will be Initially Loaded
into the File          (0-1Giga)[-] : 100000
②(Fast_Convert NoFast_Convert RMS_Puts)
Initial File Load Method (Keyword)[Fast] : FAST
③Number of Additional Records to be Added After
the Initial File Load  (0-1Giga)[0] : 0
④Will Additional Records Typically be Added in
Order by Ascending Primary Key (YES/NO)[NO] : NO
⑤Will Added Records be Distributed Evenly over the
Initial Range of Pri Key Values (YES/NO)[NO] : NO
⑥Key 0 Load Fill Percent (50-100)[100] : 100
⑦(Fixed Variable)
Record Format          (Keyword)[Var] : VARIABLE
⑧Mean Record Size      (1-32229)[-] : 80
⑨Maximum Record Size   (0,80-32229)[0] : 0

⑩(Bin2 Bin4 Bin8 Int2 Int4 Int8 Decimal String
   Dbin2 Dbin4 Dbin8 Dint2 Dint4 Dint8 Ddecimal Dstring)
Key 0 Data Type      (Keyword)[Str] : STRING
⑪Key 0 Segmentation desired (Yes/No)[No] : NO
⑫Key 0 Length         (1-255)[-] : 9
⑬Key 0 Position        (0-32220)[0] : 0
⑭Key 0 Duplicates allowed (Yes/No)[No] : NO
⑮File Prolog Version    (0-3)[3] : 3

```

(Continued on next page)

Example 4-1 (Cont.) Sample FDL Edit Session

```

①Key 0 Bucket Size          (1-63)[3]      : 3
②Key 0 Name                  (1-32 chars)[null]
: SSNUM
③Global Buffers desired      (Yes/No)[No]     : NO
④The Depth of Key 0 is Estimated to be No Greater
than 2 Index levels, which is 3 Total levels.
⑤Press RETURN to continue (~Z for Main Menu)
      VAX-11 FDL Editor
Add      to insert one or more lines into the FDL definition
Delete   to delete one or more lines from the FDL definition
Exit     to leave the FDL Editor after creating the FDL file
⑥Help    to obtain information about the FDL Editor
Invoke   to initiate a script of related questions
Modify   to change existing line(s) in the FDL definition
Quit     to abort the FDL Editor with no FDL file creation
Set      to specify FDL Editor characteristics
View     to display the current FDL Definition
⑦Main Editor Function        (Keyword)[Help] : EXIT
⑧DISK$: [FOI.RMS]FDL_SESSION_EXAMPLE.FDL;1 40 lines
  
```

- ① The Main Editor Function menu displays the EDIT/FDL commands.
- ② The INVOKE command displays the Script Title Selection menu. Note that HELP is the default command so if you want online help, just press the RETURN key.
- ③ The Script Title Selection menu shows the seven scripts you can choose to help you design your file. There is no default so you must explicitly select one of the scripts.
- ④ Choose the INDEXED script to design an indexed data file.
- ⑤ Choose a disk volume cluster size of 3.
- ⑥ Define only one key — the primary key.
- ⑦ This menu provides a selection of graphic display types.
- ⑧ Select a line plot display.
- ⑨ Select 100,000 records to be loaded initially.
- ⑩ Select the CONVERT/FAST_LOAD method of loading records into the data file.
- ⑪ Opt for no additional records after the initial load.

- ⑪ Indicate that records will be added by descending primary key.
- ⑫ Indicate that records will not be spread evenly over the initial range of primary key values.
- ⑬ Elect a fill level of 100 percent for the primary index buckets.
- ⑭ Choose the variable record format.
- ⑮ Select an average record size of 80 characters.
- ⑯ Select an unlimited maximum record size.
- ⑰ Select the string data type for the primary key.
- ⑱ Opt to disallow segmentation in the primary key.
- ⑲ Set the length of the primary key to 9 bytes.
- ㉑ Define the initial position of the primary key at column 0.
- ㉒ Opt to disallow duplicates of the primary key.
- ㉓ Choose the Prolog 3 version.
- ㉔ Select data key compression.
- ㉕ Select data record compression.
- ㉖ Select index compression.
- ㉗ This is a line plot showing bucket size against index depth.
- ㉘ Type "FD" to finish the design session.
- ㉙ Enter the title of your FDL file specification.
- ㉚ Enter the file specification of your data file.
- ㉛ Select the CARRIAGE RETURN carriage control.
- ㉜ This display shows the tuning emphasis you chose to design your file. It also shows suggested bucket sizes for various index level depths and other tuning information.
- ㉝ Select the default bucket size for the primary key.
- ㉞ Enter the name of the primary key.
- ㉟ Choose whether you want global buffers.
- ㊱ This message shows the depth of the primary key index and gives the total number of levels.
- ㊲ Press the RETURN key to display the main menu.

- This is the main menu.
- Use the EXIT command to exit the editor and to create the FDL file.
- This message shows the resulting FDL file specification and the number of lines it contains.

You will notice that most of the defaults suggested by the editor have been selected. There are three ways that you can select these defaults:

- Press the RETURN key without entering a value.
- Use the /RESPONSES=AUTOMATIC qualifier when you invoke EDIT /FDL.
- Use the following sequence:
 - 1 Select the SET command at the main menu level.
 - 2 Select RESPONSES at the SET menu level.
 - 3 Take the default (AUTO) when the utility prompts for "Default responses in script."

When EDIT/FDL creates an FDL file, it groups the attributes into major sections. The section headings are called *primary attributes* and the attributes within a primary section are called *secondary attributes*. Certain secondary attributes contain a third level of attributes called *qualifiers*.

The objective of using EDIT/FDL is to create an FDL file with optimized values for the various attributes. A completed FDL file is a list of the primary and secondary attributes together with related qualifier values. If a primary or secondary attribute does not appear in the FDL file, it is assigned its default value.

Example 4-2 shows an FDL file. IDENT, SYSTEM, FILE, RECORD, AREA n, and KEY n are primary attributes; the others are secondary attributes.

Example 4-2 Sample FDL File

```

IDENT  " 1-MAR-1985 14:07:46  VAX-11 FDL Editor"
SYSTEM
SOURCE          VAX/VMS
FILE
  GLOBAL_BUFFER_COUNT  0
  NAME                 DISK$RMS: [RMSTEST] INDEXED.DAT;3
  ORGANIZATION         indexed
  OWNER                [RMS1,TEST]
  PROTECTION            (system:RWED, owner:RWED, group:RE, world:)
RECORD
  BLOCK_SPAN           yes
  CARRIAGE_CONTROL      none
  FORMAT               variable
  SIZE                 2048
AREA 0
  ALLOCATION             233
  BEST_TRY_CONTIGUOUS   yes
  BUCKET_SIZE           5
  EXTENSION             60
  
```

(Continued on next page)

Example 4-2 (Cont.) Sample FDL File

```
AREA 1
  ALLOCATION          5
  BEST_TRY_CONTIGUOUS yes
  BUCKET_SIZE        5
  EXTENSION           5

AREA 2
  ALLOCATION          18
  BEST_TRY_CONTIGUOUS yes
  BUCKET_SIZE         3
  EXTENSION           6

KEY 0
  CHANGES            no
  DATA_AREA          0
  DATA_FILL          100
  DATA_KEY_COMPRESSION no
  DATA_RECORD_COMPRESSION no
  DUPLICATES          no
  INDEX_AREA          1
  INDEX_COMPRESSION   no
  INDEX_FILL          100
  LEVEL1_INDEX_AREA   1
  NAME                 "NUM"
  NULL_KEY            no
  PROLOG              3
  SEGO_LENGTH         8
  SEGO_POSITION        0
  TYPE                bin8

KEY 1
  CHANGES            yes
  DATA_AREA          2
  DATA_FILL          100
  DATA_KEY_COMPRESSION yes
  DUPLICATES          yes
  INDEX_AREA          2
  INDEX_COMPRESSION   yes
  INDEX_FILL          100
  LEVEL1_INDEX_AREA   2
  NAME                 "NAME"
  NULL_KEY            yes
  NULL_VALUE          0
  SEGO_LENGTH         39
  SEGO_POSITION        9
  TYPE                string
```

4.1.2.2 Designing an FDL File

When you want to create an FDL file, you invoke EDIT/FDL with a DCL command in the following form:

```
EDIT/FDL/CREATE fdl-file-spec
```

The /CREATE qualifier tells EDIT/FDL that you want to create an FDL file with the name specified by the fdl-file-spec parameter. When the utility displays the main menu, select the INVOKE command. In response to the INVOKE command, EDIT/FDL prompts you for a script and the only scripts appropriate to creating a file are INDEXED, RELATIVE, and SEQUENTIAL.

You can get to this point in a session directly by specifying the /SCRIPT qualifier on the DCL command line. For example, to create an indexed FDL file, you would enter this command:

```
EDIT/FDL/CREATE/SCRIPT=INDEXED fdl-file-spec
```

When you select the script, EDIT/FDL prompts you for information about the data file you want to create. Each prompt consists of a short question, a list or a range of acceptable values in parentheses, the required type of the value in parentheses, and the default answer in brackets. For example, one of the questions in the INDEXED script looks like this:

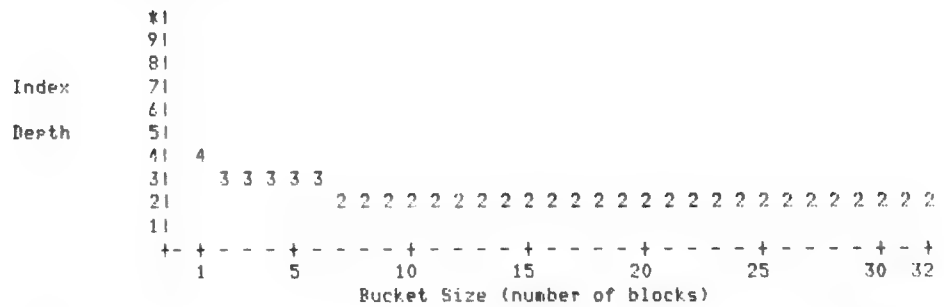
```
Number of Keys to Define (1-255)[1] :
```

EDIT/FDL prompts you for the number of keys you want to use with your indexed data file. The default response is 1, the required primary key. To take the default, press the RETURN key.

When you see the symbol [-], no default is allowed and you must respond with an acceptable value, usually shown within parentheses.

If you specify either the SEQUENTIAL or RELATIVE scripts, EDIT/FDL returns you to the main menu when you complete the dialog. If you specify the INDEXED script, one of the questions EDIT/FDL asks is your choice of design graphics mode — Line_Plot or Surface_Plot. When you have completed the dialog, EDIT/FDL displays the selected graph to help you make your final design choices.

The Line_Plot graph plots bucket size against index depth. All things being equal, the size of the buckets determines the number of levels in the index, and the number of levels has a direct effect on the run-time performance of an indexed file. Fewer levels generally decreases the average number of keys searched while the index tree is traversed. However, flatter files contain more records per data bucket and may cause longer data bucket search times. So the Line_Plot graph is a valuable tool to help you decide on the best bucket size for your application. Figure 4-1 shows a Line_Plot graph.

Figure 4-1 A Line_Plot Graph

ZK-980-82

As shown in Figure 4-1, a bucket size of 1 block results in an index with 5 levels. Increasing the bucket size to 2 blocks reduces the number of index levels to 4, but an increase to 5 blocks does not reduce the number of index levels at all. A bucket size of 7 blocks, however, reduces the number of index levels to 3.

When you choose the bucket size, remember that the graphs do not display the data level. For example, if you want 3 levels in the file, then you must limit the number of index levels to 2.

The Surface_Plot graphics mode lets you choose a range of values in order to see their effect. EDIT/FDL prompts you to enter a lower and upper bound for one of the following values:

- Load fill percent
- Key length
- Record size
- Initial load record count
- Additional record count

The selected range is displayed along the graph's Y axis.

The variable on the graph's X axis is bucket size. The numbers in the field portion of the graph indicate the number of levels at each bucket size for each of the other values.

Figure 4-2 is a Surface_Plot graph that shows a range of values for initial fill factor from 100% to 40%.

7K.949.82

Figure 4-3 shows the attributes that you can alter when EDIT/FDL displays the graph. Note that each attribute has a 2-letter mnemonic. To alter an attribute, you specify the corresponding mnemonic. To refresh the display, press the RETURN key. To begin the final design phase, enter "FD."

Figure 4-3 Design Mnemonics

```

PV-Prologue Version      3 KT-Key 0   Type      Strind FD-Final Design Phase
DK-Dup Key 0   Values  No KL-Key 0   Length    10 KP-Key 0   Position    0
RC-Data Record Comp    0% KC-Data Key Comp    0% IC-Index Record Comp    0%
RF-Record Format Variable RS-Mean Record Size    256
LM-Load Method Fast_Conv IL-Initial Load    50000 AR-Added Records    5000
Which File Parameter      (Mnemonic)[refresh]      ;

```

ZK-950-82

During the final design phase, EDIT/FDL gives you an opportunity to supply values for such attributes as TITLE, an optional primary that lets you label the FDL file. (Most of these questions are also applicable to designing sequential and relative files.) When you have answered the questions, EDIT/FDL assigns the values to the FDL attributes. Then you can then return to the main menu to display the resulting FDL file.

At this point, the script is complete. To create the displayed FDL file, you give the EXIT command. To abort the session without creating an FDL file, give the QUIT command.

You can use the ADD command to assign values to any attribute the script omitted. Remember that if an attribute does not appear in the FDL file, it takes its default value. (For a list of the default values for each attribute, see the *VAX/VMS File Definition Language Facility Reference Manual*. To modify an attribute, use the MODIFY command; to add an attribute, use the ADD command; and to delete an attribute, use the DELETE command.

4.1.2.3 Setting Characteristics for FDL Files

The FDL file that you use to create data files has certain characteristics of its own. For instance, by default FDL files have variable-length records and the default carriage control for FDL files is carriage return. Other values such as protection come from the VAX RMS system and process defaults.

You can create an FDL file with characteristics other than the defaults by first creating an FDL file to specify the characteristics you want to apply to the other FDL files. Then, use DCL to assign this FDL file the logical name EDF\$MAKE_FDL. The Edit/FDL Utility will use this file to determine the file characteristics for any FDL files it subsequently creates.

Note that the contents of the FDL files are still determined by the attribute values assigned during the EDIT/FDL session; only the external characteristics of the FDL files are affected by the EDF\$MAKE_FDL file.

For example, use the following procedure to customize the protection for your FDL files.

- 1 Create the FDL file OUTSPEC.FDL containing only these attributes:

```
FILE
    PROTECTION (SYSTEM=RWED, OWNER=RWED, GROUP=R)
```

- 2 Assign this file the logical name EDF\$MAKE_FDL with the following DCL command:

```
ASSIGN OUTSPEC.FDL EDF$MAKE_FDL
```

From this point on, FDL files created by the Edit/FDL Utility would allow RWED access to SYSTEM and to OWNER, would allow READ access only to GROUP members, and would deny access to WORLD.

4.1.3 Using the FDL Routines

You can also define file-creation characteristics with the FDL callable utility routines. The FDL routines provide you with the functions of the File Definition Language, and they allow you to set file creation characteristics from within your application.

There are four FDL routines:

FDL\$CREATE	Creates a file from an FDL specification and then closes the file. See Section 4.2.4 for more information.
FDL\$GENERATE	Produces an FDL specification by interpreting a set of VAX RMS control blocks. It then writes the FDL specification either to an FDL file or to a character string.
FDL\$PARSE	Parses an FDL specification, allocates control blocks, and then fills in the relevant fields.
FDL\$RELEASE	Deallocates the virtual memory used by the VAX RMS control blocks created by FDL\$PARSE. You must use FDL\$PARSE to fill in (populate) the control blocks if you plan to release the memory with FDL\$RELEASE later.

Because the FDL\$GENERATE, FDL\$PARSE, and FDL\$RELEASE routines allow you to use the run-time, as well as the creation-time capabilities of VAX RMS, you must call them from a language that can access the VAX RMS control block fields that specify the CONNECT options. This may be difficult from a high-level language.

Example 4-3 shows how to call the FDL\$PARSE and FDL\$GENERATE routines from a PASCAL program.

Example 4-3 Using FDL Routines in a PASCAL Program

```

[INHERIT ('SYS$LIBRARY:STARLET')]
PROGRAM example2 (input,output,order_master);
(* This program fills in its own FAB, RAB, and *)
(* IABs by calling FDL$PARSE and then generates *)
(* an FDL specification describing them. *)
(* It requires an existing input FDL file *)
(* (TESTING.FDL) for FDL$PARSE to parse. *)
TYPE
(** *)
(* FDL CALL INTERFACE CONTROL FLAGS *)
(*- *)

$BIT1 = [BIT(1),UNSAFE] BOOLEAN;
FDL2$TYPE = RECORD CASE INTEGER OF
1: (FDL$FDLDEF_BITS : [BYTE(1)] RECORD END;
);
2: (FDL$V_SIGNAL : [POS(0)] $BIT1;
(* Signal errors; don't return *)
FDL$V_FDL_STRING : [POS(1)] $BIT1;
(* Main FDL spec is a char string *)
FDL$V_DEFAULT_STRING : [POS(2)] $BIT1;
(* Default FDL spec is a char string *)
FDL$V_FULL_OUTPUT : [POS(3)] $BIT1;
(* Produce a complete FDL spec *)
);
END;

mail_order = RECORD
order_num : [KEY(0)] INTEGER;
name : PACKED ARRAY[1..20] OF CHAR;
address : PACKED ARRAY[1..20] OF CHAR;
city : PACKED ARRAY[1..19] OF CHAR;
state : PACKED ARRAY[1..2] OF CHAR;
zip_code : [KEY(1)] PACKED ARRAY[1..5]
OF CHAR;
item_num : [KEY(2)] INTEGER;
shipping : REAL;
END;

```

(Continued on next page)

Example 4-3 (Cont.) Using FDL Routines in a PASCAL Program

```

order_file = [UNSAFE] FILE OF mail_order;
ptr_to_FAB = ^FAB$TYPE;
ptr_to_RAB = ^RAB$TYPE;
byte = 0..255;

VAR
  order_master : order_file;
  flags        : FDL2$TYPE;
  order_rec    : mail_order;
  temp_FAB     : ptr_to_FAB;
  temp_RAB     : ptr_to_RAB;
  status       : integer;

FUNCTION FDL$PARSE
  (%STDESCR FDL_FILE : PACKED ARRAY [LL.:INTEGER]
   OF CHAR;
  VAR FAB_PTR : PTR_TO_FAB;
  VAR RAB_PTR : PTR_TO_RAB) : INTEGER; EXTERN;

FUNCTION FDL$GENERATE
  (%REF FLAGS : FDL2$TYPE;
  FAB_PTR : PTR_TO_FAB;
  RAB_PTR : PTR_TO_RAB;
  %STDESCR FDL_FILE_DST : PACKED ARRAY [LL.:INTEGER]
   OF CHAR) : INTEGER;
  EXTERN;

BEGIN
  status := FDL$PARSE ('TESTING',TEMP_FAB,TEMP_RAB);
  flags::byte := 0;
  status := FDL$GENERATE (flags,
                        temp_FAB,
                        temp_RAB,
                        'SYS$OUTPUT:');

END.

```

For more information on the FDL routines, see the *VAX/VMS Utility Routines Reference Manual*.

4.2 Creating a File

After you have selected the creation characteristics for your file, you need to create the file using those characteristics. You can create a file in one of four ways:

- With the VAX RMS Create service
- With the Create/FDL Utility
- With the Convert Utility
- With the FDL\$CREATE routine

4.2.1 Using the VAX RMS Create Service

This service creates a new data file assigning it the attributes you specify in the FAB. If you have defined any XABs, those characteristics are applied to the file as well. If you set fields in the XAB that correspond to fields in the FAB, the XAB fields override the FAB fields.

When you use the Create service to create a file, the file remains open until you explicitly close it.

If you set the create-if (CIF) bit in the FOP (file-processing options) field of the FAB, you can open an existing file with the VAX RMS Create service. If the file you are trying to create has the same name as an existing file, the Create service opens the existing file instead of creating the new file.

The Create service allows you to set file-creation characteristics and to create the file directly from your application program.

For more information on the Create service, see the *VAX Record Management Services Reference Manual*.

4.2.2 Using the Create/FDL Utility

Unlike using the Create service, using FDL to create a file is more of a 2-step process. You must first create the FDL file using EDIT/FDL, and then use another VAX RMS utility or your application program to create the data file.

One of the utilities you can use to create a file is the Create/FDL Utility (CREATE/FDL). CREATE/FDL creates an empty data file from the specifications in an existing FDL file. This capability allows you to use EDIT/FDL to create standard FDL files that describe commonly needed data files and then to use CREATE/FDL to create the data files as they are needed.

For example, to create an empty data file called CUSTRECS.DAT from the specifications in an FDL file called INDEXED.FDL, you would use the following command.

```
CREATE/FDL=INDEXED.FDL CUSTRECS.DAT
```

4.2.3 Using the Convert Utility

Another VAX RMS utility that creates an output data file from the specifications in an FDL file is the Convert Utility (CONVERT). However, instead of being empty, the new output file generally contains data records from the input file unless the input file was also empty.

If you want to use CONVERT to change the characteristics of a particular file, you can use the following DCL command:

```
CONVERT/FDL=fdl-file input-file output-file
```

This command creates the new file you named with the output-file parameter and assigns it the characteristics you specified in the FDL file.

For more information on populating data files with CONVERT, see Section 4.4.

4.2.4 Using the FDL\$CREATE Routine

You can also create data files according to your specifications with the FDL\$CREATE callable utility routine. FDL\$CREATE is the FDL routine most likely to be called from a high-level language. It creates a file from an FDL specification and then closes the file.

This routine performs the same function as the Create/FDL Utility, but it allows you to create data files from your application. However, it allows you to use only the creation-time capabilities of VAX RMS.

Example 4-4 shows how to call the FDL\$CREATE routine from a FORTRAN program.

Example 4-5 shows how to call FDL\$CREATE from a COBOL program.

Example 4-4 Using the FDL\$CREATE Routine in a FORTRAN Program

```

*      This program calls the FDL$CREATE routine. It
*      creates an indexed output file named NEW_MASTER.DAT
*      from the specifications in the FDL file named
*      INDEXED.FDL. You can also supply a default file name
*      and a result name (that receives the name of the created
*      file. The program also returns all the statistics.
*
      IMPLICIT      INTEGER*4      (A - Z)
      EXTERNAL      LIB$GET_LUN,    FDL$CREATE
      CHARACTER      IN_FILE*11     /'INDEXED.FDL'/,
1      OUT_FILE*14    /'NEW_MASTER.DAT'/,
1      DEF_FILE*11    /'DEFAULT.FDL'/,
1      RES_FILE*50
      INTEGER*2      FIDBLK(3)      /0,0,0/
      I = 1
      STATUS = FDL$CREATE (IN_FILE,OUT_FILE,
                           DEF_FILE,RES_FILE,FIDBLK,,)
      IF (.NOT. STATUS) CALL LIB$STOP (XVAL(STATUS))
*
      STATUS=LIB$GET_LUN(LOG_UNIT)
      OPEN (UNIT=LOG_UNIT,FILE=RES_FILE,STATUS='OLD')
      CLOSE (UNIT=LOG_UNIT, STATUS='KEEP')
*
      WRITE (6,1000) (RES_FILE)
      WRITE (6,2000) (FIDBLK (I), I=1,3)
*
1000  FORMAT (1X,'The result file name is: ',A50)
*
2000  FORMAT (/1X,'FID-NUM: ',I5/,
1      1X,'FID-SEQ: ',I5/,
1      1X,'FID-RVN: ',I5)
*
      END

```

Example 4-5 Using the FDL\$CREATE Routine from a COBOL Program

```

*      FDLCR.COB
*
*      This program calls the FDL$CREATE routine. It creates
*      an indexed output file named NEW_MASTER.DAT from the
*      specifications in the FDL file named INDEXED.DAT. You
*      can also supply a default file name and a result name
*      (that receives the name of the created file). The
*      program also returns all the FDL$CREATE statistics.
*
*      DATA NAMES:
*
*      OUT-REC      defines the output record
*      STATVALUE    receives the status value from the routine
*                   call
*      NORMAL       receives the value from SS$NORMAL
*      FIDBLOCK     receives the FDL$CREATE statistics. There
*                   are three:
*                   (1) file identification number (FID-NUM)
*                   (2) file sequence number      (FID-SEQ)
*                   (3) relative volume number    (RVN)
*      RESNAME      receives the name of the file that is created
*                   (the result file name)
*
IDENTIFICATION DIVISION.
PROGRAM-ID. FDL-CREATE-EXAMPLE.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX-780.
OBJECT-COMPUTER. VAX-780.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT OUT-FILE ASSIGN TO 'NEWMASTER.DAT'.

DATA DIVISION.
FILE SECTION.
FD      OUT-FILE
        DATA RECORD IS OUT-REC.
01      OUT-REC.
        02      OUT-NUM      PIC X(4).
        02      OUT-NAME     PIC X(20).
        02      OUT-COLOR    PIC X(4).
        02      OUT-WEIGHT   PIC X(4).
        02      SUPPLNAME    PIC X(20).
        02      FILLER       PIC X(28).

```

(Continued on next page)

Example 4-5 (Cont.) Using the FDL\$CREATE Routine from a COBOL Program

```

WORKING-STORAGE SECTION.
01  MORE-DATA-FLAGS      PIC XXX      VALUE 'YES'.
    88  THERE-IS-DATA      VALUE 'YES'.
    88  THERE-IS-NO-DATA   VALUE 'NO '.

01  STATVALUE            PIC S9(9)     COMP.

01  FIDBLOCK             USAGE IS COMP.
    02  NUM              PIC S9(9)     VALUE 0.
    02  SEQ              PIC S9(9)     VALUE 0.
    02  RVN              PIC S9(9)     VALUE 0.

01  RESNAME              PIC X(50).

PROCEDURE DIVISION.
MAIN.
    PERFORM CREATE-FILE THRU DISPLAY-STATS.
    STOP RUN.

CREATE-FILE.
    CALL 'FDL$CREATE' USING BY DESCRIPTOR 'INDEXED.FDL'
                          BY DESCRIPTOR 'NEWMASTER.DAT'
                          BY DESCRIPTOR 'DEFAULT.DAT'
                          BY DESCRIPTOR RESNAME
                          BY REFERENCE FIDBLOCK
                          BY VALUE 0
                          BY VALUE 0
                          BY VALUE 0
                          BY VALUE 0
                          BY VALUE 0
                          GIVING STATVALUE.

    IF STATVALUE IS FAILURE
    CALL 'LIB$STOP' USING BY VALUE STATVALUE.

DISPLAY-STATS.
    DISPLAY 'The result file name is: ',RESNAME CONVERSION.
    DISPLAY 'FID number:           ',NUM CONVERSION.
    DISPLAY 'FID sequence:         ',SEQ CONVERSION.
    DISPLAY 'Volume number:        ',RVN CONVERSION.

```

4.3 Defining File Protection

After you have created a file, you want to protect it against accidental or unauthorized access. You can protect a disk file in two ways:

- User identification codes (UICs)
- Access control lists (ACLs)

Tape files can be protected only with UICs.

4.3.1 UIC-Based Protection

You can protect both disk and magnetic tape files with UICs. This type of protection is made up of two parts: an owner UIC and a protection code.

The owner UIC is normally the UIC of the person who created the file. The protection code indicates who is allowed access and what type of access they are permitted.

When you try to open a file, your UIC is compared to the owner UIC of the file. Depending on the relationship of the UICs, you may be classified under one or more of the following categories:

- SYSTEM
- OWNER
- GROUP
- WORLD

Depending on your classification, you may be allowed or denied the following types of access:

READ	Can examine, print, or copy a disk or tape file
WRITE	Can modify or write to a disk or tape file
EXECUTE	Can execute a disk file that contains executable program images
DELETE	Can delete a disk file

You can specify the UIC-based protection value you need when the file is created if you use either an FDL specification or if you use VAX RMS directly.

After you create a file, you can change its UIC-based protection with the DCL command SET PROTECTION. For more information on this command, see the *VAX/VMS DCL Dictionary*.

The previous list omits *CONTROL access* because it is never specified in the standard UIC-based protection code. However, CONTROL access can be specified in an ACL and is automatically granted to certain user categories when UIC-based protection is evaluated.

CONTROL access grants the accessor all the privileges of the object's actual owner. For more information, see the *Guide to VAX/VMS System Security*.

4.3.2 ACL-Based Protection

You can also protect disk files with access control lists. (ACLs cannot be used with files on magnetic tape.)

An ACL is a list of entries that grant or deny access to a specific person or group for a particular file. ACLs offer more scope than UICs in determining what action you want taken when someone tries to access your file. You can provide an ACL on any file to permit as much or as little access as you want in each case.

You can specify the ACL for a file when you create it if you use VAX RMS directly. You cannot specify an ACL in an FDL specification and ACLs are not supported over DECnet.

After it is created, you can define the access control list for your file with the ACL Editor. You can invoke this editor with either of the following DCL commands:

- EDIT/ACL
- SET FILE/ACL

For more information on how to invoke, modify, and display ACLs, see the *VAX/VMS Access Control List Editor Reference Manual*. For additional information on VAX/VMS security features, see your system or security manager, or consult the *Guide to VAX/VMS System Security*.

4.4 Populating a File

The following two sections explain how to populate files using the Convert Utility.

4.4.1 Using the Convert Utility

The Convert Utility lets you create and populate a file.

To create a file, you need an input data file and an FDL file that describes the output file you want to create. When you have these, you issue a CONVERT command in the following form:

```
CONVERT/CREATE/FDL=fdl-file input-file output-file
```

As with the CREATE/FDL command, this command creates a file with the name specified by the output-file parameter and with characteristics specified in your FDL file. Unlike the CREATE/FDL command, CONVERT populates the output file with the records from the input file.

For example, to create the file CUST.IDX from the specifications in the FDL file STDINDEX.FDL and copy the records from the input file CUST.SEQ into CUST.IDX, you would issue the following command:

```
CONVERT/CREATE/FDL=STDINDEX.FDL CUST.SEQ CUST.IDX
```

The records in CUST.IDX are assigned the characteristics specified in the file STDINDEX.FDL.

4.4.2 Using the Convert Routines

You can invoke the functions of the Convert Utility from your application program by calling a series of convert routines.

CONVPASS_FILES	Names the files to be converted. You can also specify an FDL file.
CONVPASS_OPTIONS	Indicates the CONVERT qualifiers that you want to use. You may specify any legal CONVERT option, or you may simply take the defaults.
CONVCONVERT	Copies records from one or more source data files to a second output data file. The output file does not have to have the same file organization and format as the first.

The routines must be called in this order.

Example 4-6 shows how to call the CONVERT routines from a FORTRAN program.

Example 4-6 Using the CONVERT Routines in a FORTRAN Program

```

*      This program calls the routines that perform the
*      functions of the Convert Utility. It creates an
*      indexed output file named CUSTDATA.DAT from the
*      specifications in an FDL file named INDEXED.FDL.
*      The program then loads CUSTDATA.DAT with records
*      from the sequential file SEQ.DAT. No exception
*      file is created. This program also returns all of
*      the CONVERT statistics.
*      Program declarations
      IMPLICIT      INTEGER*4 (A - Z)

*      Set up parameter list: number of options, CREATE,
*      NOSHARE, FAST_LOAD, MERGE, APPEND, SORT, WORK_FILES,
*      KEY=0, NOPAD, PAD CHARACTER, NOTRUNCATE,
*      NOEXIT, NOFIXED_CONTROL, FILL_BUCKETS, NOREAD_CHECK,
*      NOWRITE_CHECK, FDL, and NOEXCEPTION.
*
      INTEGER*4      OPTIONS(19),
1 /18,1,0,1,0,0,1,2,0,0,0,0,0,0,0,0,0,1,0/

*      Set up statistics list as an array with the
*      number of statistics that you want. There are four
*      --- number of files, number of records, exception
*      records, and good records, in that order.

```

(Continued on next page)

Example 4-6 (Cont.) Using the CONVERT Routines in a FORTRAN Program

```

      INTEGER*4      STATSBLK(5) /4,0,0,0,0/
*      Declare the file names
      CHARACTER      IN_FILE*7 /'SEQ.DAT'/.
      1              OUT_FILE*12 /'CUSTDATA.DAT'/.
      1              FDL_FILE*11 /'INDEXED.FDL'/
*      Call the routines in their required order.
      STATUS = CONV$PASS_FILES (IN_FILE, OUT_FILE, FDL_FILE)
      IF (.NOT. STATUS) CALL LIB$STOP (XVAL(STATUS))
      STATUS = CONV$PASS_OPTIONS (OPTIONS)
      IF (.NOT. STATUS) CALL LIB$STOP (XVAL(STATUS))
      STATUS = CONV$CONVERT (STATSBLK)
      IF (.NOT. STATUS) CALL LIB$STOP (XVAL(STATUS))
*      Display the statistics information.
      WRITE (6,1000) (STATSBLK(I),I=2,5)
1000  FORMAT (1X,'Number of files processed: ',I5/,
      1      1X,'Number of records: ',I5/,
      1      1X,'Number of exception records: ',I5/,
      1      1X,'Number of valid records: ',I5)
      END

```

Example 4-7 shows how to call the CONVERT routines from a COBOL program.

Example 4-7 Using the CONVERT Routines in COBOL Program

```

*      CONV.COB
*
*      This program calls the routines that perform the
*      functions of the Convert Utility. It creates an
*      indexed output file named CUSTDATA.DAT from the
*      specifications in an FDL file named INDEXED.FDL.
*      The program then loads CUSTDATA.DAT with records
*      from the sequential file SEQ.DAT. No exception
*      file is created. This program also returns all
*      of the CONVERT statistics.
*
*      DATA NAMES:
*
*      IN-REC      defines the input record
*      OUT-REC     defines the output record
*      STATVALUE   receives the status value from the
*                  routine call
*      NORMAL      receives the value from SS$_NORMAL
*      OPTIONS     defines the CONVERT parameter list
*      STATSBLK    receives the CONVERT statistics. The
*                  first data field (NUM-STATS) contains
*                  the total number of statistics you want
*                  returned. There are four:
*                  (1) number of files processed (NUM-STATS)
*                  (2) number of records processed (NUM-FILES)
*                  (3) number of exception records (NUM-RECS)
*                  (4) number of valid records (NUM-VALRECS)
*
IDENTIFICATION DIVISION.
PROGRAM-ID. PARTS.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. VAX-780.
OBJECT-COMPUTER. VAX-780.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
        SELECT IN-FILE ASSIGN TO SEQ.
        SELECT OUT-FILE ASSIGN TO CUSTDATA.

```

(Continued on next page)

Example 4-7 (Cont.) Using the CONVERT Routines in COBOL Program

```

DATA DIVISION.
FILE SECTION.
FD      IN-FILE
        DATA RECORD IS IN-REC.
01      IN-REC.
        02      IN-NUM          PIC X(4).
        02      IN-NAME        PIC X(20).
        02      IN-COLOR       PIC X(4).
        02      IN-WEIGHT      PIC X(4).
        02      SUPL-NAME      PIC X(20).
        02      FILLER        PIC X(28).

FD      OUT-FILE
        DATA RECORD IS OUT-REC.
01      OUT-REC.
        02      OUT-NUM        PIC X(4).
        02      OUT-NAME       PIC X(20).
        02      OUT-COLR      PIC X(4).
        02      OUT-WGHT      PIC X(4).
        02      SUPLNAME      PIC X(20).

WORKING-STORAGE SECTION.
01      MORE-DATA-FLAGS      PIC X(3)      VALUE 'YES'.
        88      THERE-IS-DATA      VALUE 'YES'.
        88      THERE-IS-NO-DATA    VALUE 'NO '.

01      STATVALUE            PIC S9(9)      COMP.

01      OPTIONS              USAGE IS COMP.
        02      NUM-OPTS      PIC S9(9)      VALUE 18.
        02      CREATE        PIC S9(9)      VALUE 1.
        02      NOSHARE       PIC S9(9)      VALUE 0.
        02      FASTLOAD      PIC S9(9)      VALUE 1.
        02      NOMERGE       PIC S9(9)      VALUE 0.
        02      NOPPEND       PIC S9(9)      VALUE 0.
        02      XSORT         PIC S9(9)      VALUE 1.
        02      IWORKFILES    PIC S9(9)      VALUE 2.
        02      KEYS          PIC S9(9)      VALUE 0.
        02      NOPAD         PIC S9(9)      VALUE 0.
        02      PADCHAR       PIC S9(9)      VALUE 0.
        02      NOTRUNCATE    PIC S9(9)      VALUE 0.
        02      NOEXIT        PIC S9(9)      VALUE 0.
        02      NOFIXEDCTRL   PIC S9(9)      VALUE 0.
        02      NOFILLBUCKETS PIC S9(9)      VALUE 0.
        02      NOREADCHECK   PIC S9(9)      VALUE 0.
        02      NOWRITECHECK  PIC S9(9)      VALUE 0.
        02      FDL           PIC S9(9)      VALUE 1.
        02      NOEXCEPTION   PIC S9(9)      VALUE 0.

```

(Continued on next page)

Example 4-7 (Cont.) Using the CONVERT Routines in COBOL Program

```

01  STATSBLK          USAGE IS COMP.
   02  NUM-STATS      PIC S9(9)    VALUE 4.
   02  NUM-FILES      PIC S9(9)    VALUE 0.
   02  NUM-RECS       PIC S9(9)    VALUE 0.
   02  NUM-EXCS       PIC S9(9)    VALUE 0.
   02  NUM-VALRECS    PIC S9(9)    VALUE 0.

PROCEDURE DIVISION.
MAIN.
    PERFORM CONVERT-FILE THRU DISPLAY-STATS.
    OPEN INPUT IN-FILE.
    READ IN-FILE
        AT END MOVE 'NO ' TO MORE-DATA-FLAGS.
    CLOSE IN-FILE.
    STOP RUN.

CONVERT-FILE.
    CALL 'CONV$PASS_FILES' USING BY DESCRIPTOR 'SEQ.DAT'
                                BY DESCRIPTOR 'CUSTDATA.DAT'
                                BY DESCRIPTOR 'INDEXED.FDL'
                                GIVING STATVALUE.

    IF STATVALUE IS FAILURE
        CALL 'LIB$STOP' USING BY VALUE STATVALUE.
    CALL 'CONV$PASS_OPTIONS' USING BY CONTENT OPTIONS
                                GIVING STATVALUE.

    IF STATVALUE IS FAILURE
        CALL 'LIB$STOP' USING BY VALUE STATVALUE.
    CALL 'CONV$CONVERT' USING BY REFERENCE STATSBLK
                                GIVING STATVALUE.

    IF STATVALUE IS FAILURE
        CALL 'LIB$STOP' USING BY VALUE STATVALUE.

DISPLAY-STATS.
    DISPLAY 'Number of files processed:  ',NUM-FILES CONVERSION.
    DISPLAY 'Number of records:         ',NUM-RECS CONVERSION.
    DISPLAY 'Number of exception records:',NUM-EXCS CONVERSION.
    DISPLAY 'Number of valid records:    ',NUM-VALRECS CONVERSION.

```

For more information about calling the Convert routines, see the *VAX/VMS Utility Routines Reference Manual*.

4.5 Summary of File Creation Options

This section summarizes the file-creation options that are available using VAX RMS. These options may be available as qualifiers or keywords to the OPEN statement and include aspects of file creation, such as file disposition, file characteristics, and file allocation and positioning. In addition, because the file is opened as part of a file creation operation, refer to Chapter 9 for file open run-time options associated with file creation; these are not repeated below.

4.5.1 File Disposition Options

The creation-time options below apply to file disposition.

Name of Option	Description
Create-if	Creates the file if a file of the same file name does not exist in the same directory or opens the file if a file by the same file name exists in the same directory. FDL: FILE CREATE_IF VAX RMS: FAB\$L_FOP FAB\$V_CIF
Maximize version	Creates the file with the specified version number or a version number one greater than a file of the same name in that directory. FDL: FILE MAXIMIZE_VERSION VAX RMS: FAB\$L_FOP FAB\$V_MXV
Supersede version	Supersedes the file of the same file name, file type, and version number. The file created will replace a file with the same of the same name, type, and version number in that directory. FDL: FILE SUPERSEDE VAX RMS: FAB\$L_FOP FAB\$V_SUP
Temporary, delete	Creates a temporary file (has no directory entry) marked for deletion. The file is deleted automatically when the file is closed. FDL: FILE TEMPORARY VAX RMS: FAB\$L_FOP FAB\$V_TMD

Name of Option	Description
Temporary	Creates a temporary file (has no directory entry) that is retained when the file is closed. The file can only be subsequently accessed if its internal file identifier is known (requires the use of a name block). FDL: FILE DIRECTORY_ENTRY NO VAX RMS: FAB\$L_FOP FAB\$V_TMP

4.5.2 File Characteristics

The creation-time options that define file characteristics are described below.

Name of Option	Description
Bucket size	Defines the number of blocks to be used in each bucket (unit of I/O) throughout the life of this file. This file characteristic applies only to relative and indexed files. FDL: FILE BUCKET_SIZE VAX RMS: FAB\$B_BKS or XAB\$B_BKZ
Block size	Defines the number of bytes to be used in each block (unit of I/O) throughout the life of this file. This file characteristic applies only to magnetic tape files. FDL: FILE MT_BLOCK_SIZE VAX RMS: FAB\$B_BLS
Date information	Specifies the date-time values for file backup, file creation, file expiration, and file revision. Also can set the number of file revisions. FDL: DATE attributes and FILE REVISION VAX RMS: Date and Time XAB fields
File organization	Defines the file organization: sequential, relative, or indexed. FDL: FILE ORGANIZATION VAX RMS: FAB\$B_ORG

Name of Option	Description
File protection	<p>Defines the file protection for the file being created.</p> <p>FDL: FILE OWNER, FILE PROTECTION, FILE MT_PROTECTION</p> <p>VAX RMS: Protection XAB fields</p>
Fixed control size	<p>Defines the number of bytes of the fixed control area portion of a variable with fixed control (VFC) records.</p> <p>FDL: FILE CONTROL_FIELD_SIZE</p> <p>VAX RMS: FAB\$B_FSZ</p>
Key characteristics	<p>Defines the characteristics of a key in an indexed file, including key size(s), starting position(s), key type, bucket fill size, and key options.</p> <p>FDL: KEY attributes</p> <p>VAX RMS: Key Definition XAB fields</p>
Maximum record number	<p>Defines the maximum number of records for the file. Applies only to relative files.</p> <p>FDL: FILE MAX_RECORD_NUMBER</p> <p>VAX RMS: FAB\$L_MRN</p>
Maximum record size	<p>Defines the maximum record size for all records in the file. This indicates the size of all records in a file with fixed-length records, the size of the largest record with variable-length records, or the size of the variable portion of variable with fixed control (VFC) information records. A value of 0 with variable-length records means that there is no limit on the record size, except for magnetic tape files, for which a value of 0 sets an effective maximum record size equal to the block size minus 4. Variable or VFC records must conform to certain physical limitations (see the <i>VAX Record Management Services Reference Manual</i>).</p> <p>FDL: RECORD SIZE</p> <p>VAX RMS: FAB\$L_MRS</p>

Name of Option	Description
Record attributes	<p>Defines the type of record control information associated with each record. Records can be prevented from crossing block boundaries (FDL attribute RECORD_BLOCK_SPAN) and can have one of the following carriage control conventions:</p> <ul style="list-style-type: none"> Each record is preceded by a line feed and terminated by a carriage return. Each record can contain a FORTRAN carriage control character. Each record can be in print format where the 2-byte fixed control portion. (VFC record format) of each record contains the carriage control specification. <p>FDL: RECORD_BLOCK_SPAN or RECORD CARRIAGE_CONTROL</p> <p>VAX RMS: FAB\$B_RAT</p>
Record format	<p>Defines the record format: fixed-length records, variable-length records, variable with fixed control, stream formats (delimited by certain control characters), and undefined (sequential files only).</p> <p>FDL: RECORD_FORMAT</p> <p>VAX RMS: FAB\$B_RFM</p>

4.5.3 File Allocation and Positioning

You can specify certain file allocation and positioning options with either the FAB control block or an Allocation XAB (XABALL) control block; any value specified in the Allocation XAB takes precedence over any corresponding value in the FAB. The creation-time options described below apply to file allocation and positioning.

Name of Option	Description
Allocation quantity	<p>Allocates the file or area using the number of blocks specified by this value, rounded up to the nearest even multiple of the volume's cluster size.</p> <p>FDL: FILE ALLOCATION or AREA ALLOCATION</p> <p>VAX RMS: FAB\$_ALQ or XAB\$_ALQ</p>
Areas	<p>The file can be allocated using single or multiple areas. Applies only to indexed files; sequential and relative files are always contained in a single area. Indexed files can be placed in specific areas, for example, to separate the data area from the index area.</p> <p>FDL: AREA number</p> <p>VAX RMS: XAB\$_AID</p>
Contiguous	<p>Allocates the file or area using a single extent. In other words, if the disk's unallocated space does not permit the file to be allocated contiguously, an error is returned.</p> <p>FDL: FILE CONTIGUOUS or AREA CONTIGUOUS</p> <p>VAX RMS: FAB\$_FOP FAB\$_CTG or XAB\$_AOP XAB\$_CTG</p>
Contiguous best try	<p>Attempts to allocate the file or area using a minimum number of extents. If the file cannot be allocated contiguously, an error is not returned.</p> <p>FDL: FILE BEST_TRY_CONTIGUOUS or AREA BEST_TRY_CONTIGUOUS</p> <p>VAX RMS: FAB\$_FOP FAB\$_CBT or XAB\$_AOP XAB\$_CBT</p>
Cylinder boundary	<p>Indicates that the file or area should be allocated beginning on any available cylinder boundary.</p> <p>FDL: AREA POSITION ANY_CYLINDER</p> <p>VAX RMS: XAB\$_AOP XAB\$_ONC</p>

Name of Option	Description
Cylinder position	<p>Indicates that the file or area should be positioned beginning at the specified cylinder number.</p> <p>FDL: AREA POSITION CYLINDER</p> <p>VAX RMS: XAB\$B_ALN XAB\$V_CYL and XAB\$L_LOC</p>
Default extension	<p>Defines the minimum number of blocks for a file extension (extent) when additional disk space is needed.</p> <p>FDL: FILE EXTENSION</p> <p>VAX RMS: FAB\$W_DEQ or XAB\$W_DEQ</p>
Hard positioning	<p>Indicates that VAX RMS is to return an error if the requested file or area positioning or alignment cannot be performed.</p> <p>FDL: AREA EXACT_POSITIONING</p> <p>VAX RMS: XAB\$B_AOP XAB\$V_HRD</p>
Logical block position	<p>Indicates the file or area should be positioned beginning at the specified logical block number.</p> <p>FDL: AREA POSITION LOGICAL</p> <p>VAX RMS: XAB\$B_ALN XAB\$V_LBN and XAB\$L_LOC</p>
Related file position	<p>Indicates that the file or area should be positioned as close as possible to a related file, at the specified virtual block number.</p> <p>FDL: AREA POSITION FILE_ID or AREA POSITON FILE_NAME</p> <p>VAX RMS: XAB\$B_ALN XAB\$V_RFI and XAB\$L_LOC</p>
Virtual block position	<p>Indicates the file or area should be positioned beginning at the specified virtual block number.</p> <p>FDL: AREA POSITION VIRTUAL</p> <p>VAX RMS: XAB\$B_ALN XAB\$V_VBN and XAB\$L_LOC</p>
Volume number	<p>Indicates the volume number of the volume set where the file or area will be placed when it is created.</p> <p>FDL: AREA VOLUME</p> <p>VAX RMS: XAB\$W_VOL</p>

Creating and Populating Files

For the list of the run-time options that are common to creating and opening a file, see Chapter 9.

For more information on the options listed above, see Chapter 2. For more detailed information on the programming aspects of these options, see the *VAX Record Management Services Reference Manual*.

5

Locating and Naming Files

When creating or opening a file, your program must provide the appropriate file specification. Typically, VAX languages require a file specification argument for an OPEN statement that names a file being created or locates a file being opened.

There are a number of ways to locate a file using a VAX/VMS file specification. The most direct way is to provide the actual complete file specification, which is often used when creating a new file. Or, you might let your program supply the defaults so the user need only enter the file name component of a file specification to access the file.

Unlike small computer systems which might have only one mass storage device, a VAX system may have many disk and magnetic tape devices. To eliminate the need to always specify the device and directory in which a file resides, VAX RMS uses the current device and directory position of your process as defaults.

5.1 Understanding File Specifications

A file specification on a VAX/VMS system consists of up to six components, several of which have defaults when they are not specified. To allow VAX RMS to identify the boundaries of each component, certain characters separate the components in a file specification. These characters mark the beginning or the end of a file specification component and must be supplied if a subsequent component is present. This is the general format of a complete file specification:

`node::device:[directory-name]filename.type;version`

The following table lists the characters that separate (begin or end) each component of a file specification.

Component	Separator Character(s)
Node	Double colon (::) ends a node name.
Device	Single colon (:) ends a device name.
Directory	Square brackets ([]) or angle brackets (< >) delimit the directory name. Within the directory component, a period (.) separates each directory and subdirectory name.
File name	Period (.) that begins the type component also ends the file name.
Type	Period begins the type component and a semicolon (;) or period (.) ends the type component.

Some examples of valid file specifications follow:

```
DISK1:[MYDIR]FILE.DAT
[MYDIR]FILE.DAT
FILE.DAT;10
NODE::DISK5:[REMOTE.ACCESS]FILE.DAT
```

The maximum permissible length of a file specification string is 255 characters, including all separator characters. The following table lists the length limits for each of the component parts of a file specification. Note that although the collective limits exceed 255 characters, the overriding limitation is on the length of the file specification. For example, if you used the maximum number of characters allowed for a logical device name (255 characters), you could not specify any other file specification component because the length of the file specification string would exceed the 255-character limit.

Component	Number of Characters
Node	Up to 59 characters including an access control string (physical node names are up to 6 characters; logical node names are up to 15 characters)
Device	Up to 15 characters for a physical device name; up to 255 characters for a logical device name
Directory-name	Up to 39 characters for each directory and subdirectory name present
File name	Up to 39 characters
Type	Up to 39 characters
Version	Up to 5 digits, which optionally may be preceded by a hyphen (-)

Care should be taken when naming files that will be copied or accessed by remote systems. The file-naming abilities of a VAX/VMS Version 4 system exceed those of most other computer systems, including VAX systems running VAX/VMS Version 3. For example, a system running VAX/VMS Version 3 will return a syntax error when a file specification contains any of the following:

- A file name or a directory name longer than 9 characters
- A file type longer than 3 characters
- A dollar sign (\$), an underscore (_) or a hyphen (-).

Here are some examples of file specifications that are valid on a VAX/VMS Version 4 but are not valid on a VAX/VMS Version 3 system:

```
NODE::DBA2:[YOUR_DIR]FILE.DAT
NODE::DBA2:[DIR]FILETOOLONG.DAT
NODE::DBA2:[DIR]FILE_TEST.DAT
NODE::DBA2:[DIR]FILE.DATA
```

If your application makes files with VAX/VMS Version 4 file specifications available to a remote system running under VAX/VMS Version 3, you must rename these files in order for the remote system to access them. Alternatively, you could copy these files to the remote system using valid VAX/VMS Version 3 output file specifications.

File name restrictions are generally determined by the file name capabilities of the remote systems that require access to them. These restrictions must be considered part of the overall application design when network access is required.

Note

Applications that parse file specifications using VAX/VMS Version 3 file specification conventions should be modified to use the services or routines that can parse or scan file specifications using the new extended file specifications conventions. These services and routines include the VAX RMS Parse service described in the *VAX Record Management Services Reference Manual*, the Scan String for File Specification system service described in the *VAX/VMS System Services Reference Manual*, and the LIB\$FIND_FILE and LIB\$FILE_SCAN routines described in the *VAX/VMS Run-Time Library Routines Reference Manual*.

5.1.1 File Specification Formats

The selection of a file specification format is in part dependent on whether you are confining your file activity to the local node or whether you are conducting file activity on remote nodes. For example, you need not include the node name in the file specification when trying to locate a file on the local node or VAXcluster. Conversely, to locate a file on a remote node, the name of the remote node must be present either as the physical node name or as a logical name whose translation contains the appropriate node name. A logical node name can also contain access control information used to log in at the remote system.

With regard to device names, the device can be identified using either a physical name or a logical name. You can terminate a physical device name or a logical device name with a colon and place one or more file specification components (directory name, file name, file type, and version) after it.

A logical name in the device component can translate to another logical name, a physical device name, or a physical device name with additional file specification components. The logical name can be a search list, which specifies multiple file locations where the file can be found (see Section 5.2.2.)

You need include only a device name when specifying a record-oriented device, such as a terminal. However, if you choose to include other file specification components, you must abide by the rules described previously.

A logical name can also be present as the file name component if it is the only component specified in the file specification. Refer to the *VAX/VMS DCL Dictionary* for additional information on defining logical names.

Formats for locating local and remote files are described in the remainder of this section.

Local Node

The first file specification format does not include a node name.

device: [root.] [directory-name] filename.type;version

This is the general format (file-spec) of a file specification used to locate a file on the local node or VAXcluster. All components of the file specification may need to be present when, for example, the file is being created or when some of the file specification components, such as the device component, can be provided by default.

The next format is used only for ANSI-formatted magnetic tape volumes.

device: [directory-name] "quoted-ascii-a-string".;nn

Remote Node

Here are two formats used for accessing files on remote nodes.

```
node::file-spec  
node"access-control-string"::file-spec
```

The second format includes an access control string. If an access control string is specified or if the process seeking to gain access to a remote file has a proxy login account on the remote node, the specified remote process uses its access rights to locate the file. If an access control string is not specified and a proxy account does not exist on the remote system, the local process may use the default DECnet account to locate the file.

Here is another format used to locate files on remote nodes.

```
node::"foreign-file-spec"
```

The only action VAX RMS takes with the foreign file specification is to translate the logical node name, if applicable. This format is especially useful when the remote system is not a VAX/VMS system and the file specification does not conform to VAX/VMS file specification syntax conventions. Refer to the *VAX/VMS Networking Manual* for more information.

The next format does not specify a file directly. Instead, it specifies a task on the remote system.

```
node::"task-spec-string"
```

For more specific information about specifying a logical node name or using any of the file specification formats and their associated syntax rules, refer to the *VAX/VMS DCL Dictionary*.

Refer to the *VAX/VMS DCL Dictionary* for a list of the characters from the DEC Multinational character set that can be used in a file specification and the characters that can be used in quoted-ascii-a strings for ANSI magnetic tape files.

5.1.2 Using File Specification Defaults

When you omit file specification components (except for the node name), VAX RMS attempts to supply values for the missing components by using defaults. The file specification to which defaults are applied is called the *primary file specification*. Your program can supply default values for all primary file specification components using either the *default file specification* or the *related file specification*. The process executing the program can supply the specific default values for the device and directory components.

Where applicable, VAX RMS substitutes the translated logical name to the primary file specification before it applies defaults. After translating the primary file specification, VAX RMS applies the defaults from the default file specification and then the related file specification, if applicable. If applicable, VAX RMS then applies the process-supplied default values for the device and directory to obtain the fully-qualified file specification it uses to locate the file. If a name block is present and certain name block fields are specified, the resulting file specification is available to the program after certain VAX RMS services are invoked. (For more information on the application of defaults, refer to Section 6.1.

VAX RMS applies process defaults to the device and directory components when a file specification does not include these components. Therefore, you must explicitly specify the device and directory if you want to access a file outside of your process- specified device and directory. At login, the process device and directory is set to the value established by the system-defined logical name SYS\$LOGIN. VAX RMS obtains the current device by translating the logical name SYS\$DISK, and it maintains the current directory in your process context.

5.2 Logical Names and Parsing

VAX RMS translates any logical name present in a file specification at run time. The use of logical names can be desirable for several reasons, including program simplification, device independence, file independence and ease of use.

You can specify the file specification at compile (or assembly) time or the program can prompt the terminal user for it at run time. By specifying a logical name at compile time, you eliminate the need for a terminal input request from the program, yet maintain the flexibility of allowing the file specification to be specified just before run time.

If a logical name is used for the device name component, an alternate device (usually containing a recent backup copy of the device) can be substituted simply by changing the definition of the logical name. This can reduce or eliminate the downtime caused by media failure or scheduled preventive maintenance (device independence).

Similarly, if the current copy of a file is not available, an alternate file can often be used. To locate several files in a defined search order, you can use a search list. Wildcard characters can also be used to locate several files using one file specification but do not allow you to specify a search order.

Using a logical name to represent a longer file specification or file specification component, reduces the number of keystrokes the user must enter, which saves time and reduces the chance of error. For example, you could define a logical node name that translates to an actual node name and access control string for use when locating remote files. To keep the password a secret when you use this technique, the logical name should be defined interactively rather than in a command procedure.

5.2.1 Example Use of Logical Names

Regardless of the programming language used, you can use a logical name to provide one, some, or all components of a file specification. The following program example shows how to access a remote file, which is done exactly in the same way that you would access a local file, except for the presence of the node name in the file specification. Because logical names are used as file specifications, the terminal user or a command procedure must execute the following sequence of commands to execute the program:

```
$ DEFINE SRC TRNTO::USER:[STOCKROOM.PAPER]INVENTORY.DAT
$ DEFINE DST BOSTON::LPAO:
$ RUN TRANSFER
```

Example 5-1 is a simple FORTRAN program that transfers a remote file on node TRNTO to the line printer on node BOSTON, using the logical names SRC and DST, which must be defined for the process before the program is run.

In Example 5-1, standard I/O calls transfer the file's records from one device to another. Note the use of the VAX/VMS file specification format with a remote node name present. (If the remote node is other than VAX/VMS, the format of the file specification may differ.)

After opening the files and copying all the records, the program closes the channels, thereby terminating network operations. These operations are similar for applications in the other high-level VAX languages.

5.2.2 Types of Logical Names

When a logical name is defined, you can specify various attributes including the *concealed* attribute and the *terminal* attribute. By default, a logical name is neither concealed nor terminal.

To specify a logical name as either concealed or terminal, use the `/TRANSLATION_ATTRIBUTES` qualifier for the DCL commands `DEFINE` or `ASSIGN`.

Example 5-1 Using Logical Names for Remote File Access

```

PROGRAM TRANSFER
C
C   This program creates a sequential file with variable length
C   records from a sequential input file. The input and output
C   files are identified by the logical names SRC and DST,
C   respectively.
C
C   CHARACTER BUFFER*132
C
C   100  FORMAT (Q,A)
C   200  FORMAT (A)
C
C   Open the input and output files.
C
C   OPEN (UNIT=1,NAME='SRC',TYPE='OLD',ACCESS='SEQUENTIAL',
1      FORM='FORMATTED')
C   OPEN (UNIT=2,NAME='DST',TYPE='NEW',ACCESS='SEQUENTIAL',
1      FORM='FORMATTED',CARRIAGECONTROL='LIST',
2      RECORDTYPE='VARIABLE')
C
C   Transfer records until end-of-file or other error condition.
C
C   10  READ (1,100,END=20,ERR=20) NCHAR,BUFFER(:NCHAR)
C       WRITE (2,200) BUFFER(:NCHAR)
C       GOTO 10
C
C   Close the input and output files.
C
C   20  CLOSE (UNIT=2)
C       CLOSE (UNIT=1)
C       END

```

The concealed attribute insures that VAX RMS uses the device logical name when communicating with the user program. For example, if the device logical name does not have the concealed attribute, any file specification information returned to the user program would include the device's physical name rather than its logical name.

Using the concealed attribute for a physical device name, you can move a user's files from one physical device to another without the user being required to maintain the physical location of the files. Consider too, the mnemonic benefits derived by having VAX RMS communicate using logical names rather than physical names.

A search list is a logical name that contains more than one file specification. Typically a search list is used to search multiple file locations looking for a file. VAX RMS attempts to locate the file by using the first file specification in the search list, then the next, and so forth until the file is found or the search list is exhausted. Like other logical names, a search list is usually defined using the ASSIGN or DEFINE commands; however, in a search list logical name, the multiple file specifications (equivalence names) must be separated by commas.

Any of the equivalence names in the search list may be specified individually as terminal or concealed. Section 6.2 describes the use of search lists and wildcard characters for multiple file processing and parsing. For general information about using logical names, refer to the *VAX/VMS DCL Dictionary*.

Note

Concealed device logical names are no longer denoted by a double underscore as in VAX/VMS Version 3.0; the concealed property is now a translation attribute of a logical name. Programs that scan for the double underscore should invoke the Parse service and test the `NAM$L_FNB` field `NAM$V_CNCL_DEV` bit to determine if a logical name contains a concealed device instead of looking for a string containing a double underscore. File specifications containing the double underscore are no longer valid.

5.2.3 Introduction to File Specification Parsing

VAX RMS allows an application program to specify defaults for the device and directory components of a file specification as well as other components of a file specification. The method VAX RMS uses to apply defaults and translate any logical names present is called *file parsing*. In effect, VAX RMS merges the various default strings (after translating any logical names) to generate the file specification used to locate the file.

One of the functions of file parsing is to determine when a logical name is present and whether the file specification describes a file on the local node. If a node name is not present in the file specification (the file is located on the local system), VAX RMS first translates any logical names, applies defaults to any missing components, and then attempts to locate the file.

If a node name is present, VAX RMS does not process the file specification on the local node. Instead, it merges any program-specified defaults without translation and passes the defaulted, untranslated file specification to the file access listener (FAL) at the remote node; the operating system on the remote node interprets it.

With advanced file-parsing, a *single* file specification can be used to locate one file or multiple files. To locate a single file, multiple file locations or file names can be searched to ensure that the file is found. The multiple file locations or file names can be located in the same or in different directories, on different devices, on different nodes, or a combination thereof.

To locate multiple files instead of a single file, the multiple locations or file names can each be processed as a file using the same file specification. These advanced features are available only with the VAX/VMS operating system and involve the use of wildcard characters and search lists.

When a wildcard character or a search list is included in a file specification, the application program may need to preprocess the file specification before attempting to locate the file. A VAX RMS file service that operates on an unopened file (such as the Create and Open services) will perform the following file-parsing activity:

- Examine a file specification for validity
- Translate any logical names present
- Apply defaults
- Attempt to locate the file

If a name block is present, additional file-parsing activities can occur:

- Return the actual complete file specification used to access the file and its associated file identifier
- Return the length of each component of a file specification as well as other information about the file specification

Some file services, including the Open and Create services, cannot process a file specification that contains a wildcard character, and so must be preceded by the Parse service. (If a search list with no wildcard characters is present, the Parse service is usually not needed.) The Parse service is used to determine whether wildcard characters or search lists are present. It also initializes control block fields that are necessary to search for multiple files using the Search service. To use the Search service, a NAM (name) block must be present when the Parse service is invoked.

Alternatively, you can use the SYS\$FILESCAN system service (scan string for file specification) to scan a file specification for validity and optionally return the lengths of the individual file specification components without translating logical names or applying defaults. Two Run-Time Library routines, LIB\$FIND_FILE and LIB\$FILE_SCAN, perform functions that are similar to the Parse and the SYS\$FILESCAN services.

For more information on how VAX RMS parses a file specification, see Section 6.1. Section 6.3 discusses the use of directory specifications, including directory syntax conventions.

5.3 Using One File Specification to Locate Many Files

Five services can translate and apply defaults to a file specification to produce a fully parsed file specification: the Create, Open, Erase, Parse, and Rename services. Other file services must be preceded by one of these services to parse the file specification and, in some cases, to open the file.

If a file specification contains one or more wildcard characters, it must be preprocessed using the Parse and Search services before the file can be located. The Parse service sets certain bit values in a name block field called the file name status bits field (NAM\$L_FNB). This field can be tested to determine whether a wildcard character or a search list logical name is present. The Search service locates a file and specifies its name (without wildcard characters). If wildcard characters are present, you must first invoke the Search service before processing (opening or creating) the file; if wildcard characters are not present, the file can be processed without invoking the Search service.

To process a single file, you need to invoke the Search service only once; to process many files, invoke the Search service as many times as needed to return the next fully qualified file specification. When no more files match the file specification, the Search service returns a no-more-files-found message (RMS\$_NMF).

In summary, the Parse and Search services work together to provide a fully qualified file specification that the Search service uses to locate the file.

Your program can process a single file without using the Search service if neither the file specification nor the search list contain wildcard characters. If any of the file specifications in a search list contain wildcard characters, the Search service must be invoked before processing the file to prevent an invalid wildcard completion status error. If a wildcard character is present in the second or subsequent file specifications in a search list, VAX RMS does not set the wildcard bit in the file name status bits field.)

If the Parse and Search services will precede an Open service, an open-by-name-block operation should be performed by specifying the address of the name block in the name block address (FAB\$L_NAM) field and setting the file-processing options (FAB\$L_FOP) open-by-name-block (FAB\$V_NAM) bit option.

Wildcard characters cannot be present in the file specification when the Create service is invoked. There are cases in which the Parse and Search services might precede a Create operation. If the file-processing (FAB\$L_FOP) field create-if (FAB\$V_CIF) or supersede (FAB\$V_SUP) bit options are set, the program might invoke the Parse service to check for wildcard characters (or search lists) in the file specification. If either a search list or wildcard characters are found, the program must invoke the Search service before invoking the Create service. (The create-if option tries to open a file if it can be found in any of the search list locations; if the file is not found, it is created using the first file specification from the search list.) If these options are specified and a wildcard character is present when the Create service is invoked, the file specification is invalid; if a search list is present, the file is created using the first file specification from the search list.

You can either call these services directly from a VAX MACRO procedure (or as part of a USEROPEN or USER_ACTION routine in a high-level language) or execute the calls from VAX language subroutines or functions that call the VAX RMS services. The Parse and Search services require that a name block be present. Unless your language supports a means of setting values in a name block (and other control blocks) and invoking VAX RMS services, you should use a VAX MACRO procedure. FDL does not support the use of a name block.

In addition to a name block, you usually need a file access block and a record access block. To perform file services, a file access block (and, if needed, extended attribute blocks) must be present; to perform record services, a record access block must be present.

The following program shows how to use the LIB\$FIND_FILE routine to locate the desired file, which the terminal user enters. Because LIB\$FIND_FILE is used with the supplied arguments, the file specification may contain wildcard characters, a search list, and a search list that assumes the program will allow the use of "sticky" defaults, as in DCL command line parsing. The routine is called by the following VAX BASIC program USEROPEN option for the BASIC OPEN statement.

```
100 MAP (REC.1) SURNAME$ = 20%, REST$ = 60%
110 OPEN " " FOR OUTPUT AS FILE #1%, ORGANIZATION RELATIVE, &
      MAP REC.1, USEROPEN LOCATE
120 CLOSE #1%
130 END
```

The BASIC program allocates the control blocks before control is given to the USEROPEN routine; it also passes the address of the file access block (FAB) as the first argument and the address of the record access block (RAB) as the second argument. These arguments enable the VAX MACRO routine to obtain the control block addresses because the argument pointer points to the longword count of arguments, followed by the longword-length arguments. Because the VAX MACRO macros \$FAB and \$NAM are not used, access to the symbolic offsets defined for these control block are not available; thus, the VAX MACRO macros \$FABDEF and \$NAMDEF (and \$RABDEF) are specified to define these symbols for the USEROPEN routine.

In addition to locating the file using any valid file specification, the called routine also connects to the file requesting 15 global buffers (or as many global buffers as system resources permit). This routine is linked with the BASIC program to form the executable image. Example 5-2 shows this VAX MACRO routine.

Example 5-2 VAX MACRO Routine Called by a BASIC Useropen Program

```

.TITLE LOCATE
.PSECT DATA,WRT,NOEXE
.EXTERNAL LIB$SIGNAL,LIB$STOP,LIB$GET_INPUT,LIB$PUT_OUTPUT
.EXTERNAL STR$GET1_DX
$FABDEF                                ; Define FAB symbols
$RABDEF                                ; Define RAB symbols
;
IFILE: .BLKB 80                        ; Input file spec.
IFILED: .LONG 80                       ; File spec. descriptor
      .LONG IFILE
;
OFILED: .WORD 255                      ; File spec. descriptor
      .BYTE 14                         ; Specify character text
      .BYTE 2                         ; Specify descriptor class
OFILE: .LONG 0                        ; Addr filled in by STR$GET1_DX
;
DFILED: .ASCID /.DAT/                 ; Default file spec. descriptor
;
PROMPT: .ASCID /Enter the file spec: / ; User prompt
LOC_P: .ASCID /** NOTE: Global buffers unavailable **/ ;
NULL_P: .ASCID / /                    ; Blank line prompt
;
ARGS: .LONG 7                         ; 7 arguments
      .ADDRESS IFILE                 ; Input file spec.
      .ADDRESS OFILED                ; Output file spec.
      .ADDRESS CTEXT                 ; Context
      .ADDRESS DFILED                ; Default file spec.
      .ADDRESS NULL                  ; No related file spec.
      .ADDRESS STV_L                 ; STV field
      .ADDRESS UFLAGS                ; User flags
;
CTEXT: .LONG 0                        ; Context work area
NULL: .LONG 0                        ; No related file spec.
STV_L: .BLKL 1                       ; STV status return area
UFLAGS: .BLKL 1                      ; User flags
LEN: .BLKB 255
;
.PSECT CODE,NOWRT,EXE
.ENTRY LOCATE,"M<>"
;
      MOVL 4(AP),R6                  ; Move FAB address into R6
      MOVL 8(AP),R7                  ; Move RAB address into R7
      BISL2 #2,UFLAGS                ; Set flag for sticky defaults

```

(Continued on next page)

Example 5-2 (Cont.) VAX MACRO Routine Called by a BASIC Useropen Program

```

TERR:  PUSHAL  IFILED                ; Get input length
        PUSHAL  PROMPT              ; Prompt for input
        PUSHAL  IFILED              ; Input descriptor
        CALLS   #3, G`LIB$GET_INPUT ; Get input
        BLBC    RO,TERR              ; Retry on error
        PUSHAL  OFILED              ; Push descriptor address
        PUSHAL  LEN                  ; And length
        CALLS   #2, G`STR$GET1_DX   ; Allocate dynamic string
        BLBC    RO,ERR               ; Branch on error
        CALLG   ARGS, G`LIB$FIND_FILE ; Call RTL Find File Routine
        BLBC    RO,ERR               ; Branch on error
        BRW     OPEN                ; Skip on success
ERR:    PUSHL   STV_L                ; Signal error status
        PUSHL   RO                  ; codes
        CALLS   #2, G`LIB$SIGNAL    ; Display error
        BRW     TERR                ; Reenter file spec. on error

OPEN:   PUSHAL  OFILED              ; Display file spec.
        CALLS   #1, G`LIB$PUT_OUTPUT ; on screen
        MOVL    OFILE,R10           ; Move file spec. addr to R10
        $FAB_STORE FAB=R0,FNA=(R10),FAC=GET,-
        FNS=OFILED,SHR=<GET,MSE>    ; Set read-sharing global buff
        $OPEN   FAB=R0              ; Open the file
        BLBS    RO,CONNECT          ; Branch on success
        PUSHL   FAB$L_STV(R0)        ; Push STV and STS in reverse
        PUSHL   FAB$L_STS(R0)        ; order on stack to
        CALLS   #2, G`LIB$STOP      ; Signal error and stop

;
; This block of code attempts to Connect with global buffers if possible
; and uses local buffers if global buffer resources are not available.
; Because the global buffer value is set between the Open and Connect,
; all defaults are overwritten.
;
CONNECT:
        MOVL    #15,R9              ; R9 contains global buff count
        BRB     RETRY               ; Skip local buffer handing
LOCAL:  MOVL    #0,R9               ; Turn off global buffers
        $RAB_STORE RAB=R7,MBF=#6    ; Request 6 local buffers
        PUSHAL  LOC_P               ; Inform user
        CALLS   #1, G`LIB$PUT_OUTPUT ; No global buffers
RETRY:  $FAB_STORE FAB=R0,GBC=R9     ; Override default global buff.
        $CONNECT RAB=R7             ; Connect the record stream
        BLBC    RO,RERR              ; Branch on error
        BRW     DONE                ; On success, return

```

(Continued on next page)

Example 5-2 (Cont.) VAX MACRO Routine Called by a BASIC Useropen Program

```

RERR:  CMPL      R0,#RMS$_CRMP      ; Check if too many gl. bufs
       BNEQ      CERR              ; Quit if other error
       CMPL      #4,R0              ; Test if too few global bufs
       BLSS      LOCAL              ; Use local buffers
       SUBL2     #3,R0              ; Decrement R0 by 3
       BRW       RETRY              ; Attempt Connect again

CERR:
       PUSHL     RAB$L_STV(R7)       ; Push STV and STS in reverse
       PUSHL     RAB$L_STS(R7)       ; order on stack to
       CALLS     #2, G~LIB$STOP      ; Signal and end on error
DONE:  RET
       .END      LOCATE              ; Return to main program

```

Example 5-2 also shows the proper way to signal errors. Both the STS and STV fields of the FAB or RAB are used so that secondary completion information is displayed, if appropriate, by the LIB\$SIGNAL or LIB\$STOP routines. For more information on the LIB\$ routines shown here and other routines in the VAX/VMS Run-Time Library, see the *VAX/VMS Run-Time Library Routines Reference Manual*. The VAX MACRO program shown below in Example 5-3 invokes the Parse service, determines whether a wildcard character or search list is present, and conditionally branches to a sequence of instructions that invoke the Search service before the Open service. The resultant string is displayed after the file is opened.

Example 5-3 Using the Parse, Search, and Open Services

```

        .TITLE WILDFILE
;
; BEGIN DATA PROGRAM SECTION * * * * *
;
        .PSECT DATA,NOEXE,WRT
MY_NAM: $NAM      RSA=RES_STR,-      ; Result buffer address
                RSS=255,-           ; Result buffer size
                ESA=EXP_STR,-       ; Expanded buffer address
                ESS=255             ; Expanded buffer size
MY_FAB: $FAB      FOP=NAM,-         ; Use NAM block option
                NAM=MY_NAM,-        ; Pointer to NAM block
                FNA=INP_STR         ; Addr of file name string
EXP_STR:         ; Expanded string buffer
                .BLKB NAM$C_MAXRSS
RES_STR:         ; Resultant string buffer
                .BLKB NAM$C_MAXRSS
RES_STR_D:       ; Resultant string descriptor
                .BLKL 1
                .LONG RES_STR
INP_STR:         ; Input string buffer
                .BLKB NAM$C_MAXRSS
INP_STR_D:       ; Input string descriptor
                .LONG NAM$C_MAXRSS
                .LONG INP_STR
INP_STR_LEN:     ; Input string length
                .BLKL 1
PROMPT_D:        ; User prompt string
                .ASCID /Please enter the file specification: /
;
; BEGIN CODE PROGRAM SECTION * * * * *
;
        .PSECT CODE,EXE,NARRATE
        .ENTRY WILDFILE, ~M<>      ; Save no registers
        PUSHAB INP_STR_LEN         ; Address for string length
        PUSHAB PROMPT_D            ; Prompt string descriptor
        PUSHAB INP_STR_D           ; String buffer descriptor
        CALLS  #3,G*LIB$GET_INPUT   ; Get input string value
        BLBS  NO,MOVE              ; Branch on success
        BRW   EXIT                 ; Quit on error
;
; Store user input string and perform parse
;
MOVE:  MOVB  INP_STR_LEN, -         ; Set string size
        MY_FAB+FAB$B_FNS

```

(Continued on next page)

Example 5-3 (Cont.) Using the Parse, Search, and Open Services

```

PAR:  $PARSE FAB=MY_FAB           ; Parse filespec in MY_FAB
      BLBC  RO,F_ERR             ; Branch on error
      BBS   $NAM$V_WILDCARD,-    ;
      NAM$L_FNB+MY_NAM,WILD      ; Branch on bit set
      BBS   $NAM$V_SEARCH_LIST,- ;
      NAM$L_FNB+MY_NAM,WILD      ; Branch on bit set
      BRB   OPEN                 ; OK to open file
WILD: $SEARCH FAB=MY_FAB         ; Search for next file
      BLBC  RO,F_ERR             ; Branch on error
OPEN: $OPEN  FAB=MY_FAB          ; Open file
      BLBC  RO,F_ERR             ; Branch on error
      MOVZBL MY_NAM+NAM$B_RSL,-  ; Move resultant string
      RES_STR_D                     ; length to descriptor
      PUSHAB RES_STR_D            ; String buffer descriptor
      CALLS #1,G'LIB$PUT_OUTPUT   ; Display resultant filespec
      ; Connect and process file
      ;
      ;
      ; ... and
      $CLOSE FAB=MY_FAB          ; Close file
      BLBS  RO,EXIT              ; Branch on success
F_ERR: PUSHL MY_FAB+FAB$L_STV     ; Push STV and STS on stack
      PUSHL MY_FAB+FAB$L_STS     ; in reverse order
      CALLS #2,G'LIB$SIGNAL       ; Signal error
EXIT:  RET                      ; Exit with RO
      .END WILDFILE

```

Example 5-3 uses the VAX MACRO macros \$FAB and \$NAM that define the control blocks and specify the arguments for the Parse, Search, Open and Close services. It shows how to preprocess a file specification using the Parse and Search services. To process many files, you could add an unconditional branch instruction just before the symbolic address F_ERR to branch to the \$SEARCH macro at symbolic address WILD.

Refer to the *VAX Record Management Services Reference Manual* and the *VAX MACRO and Instruction Set Reference Volume* for more VAX MACRO RMS examples and information about using VAX MACRO.

An application may also need to process either one file or many files, depending on the file specification that the terminal user enters or the logical name that is provided (if the program uses a logical name in its file specification). Each of these cases are discussed below.

5.3.1 Processing One File

When only a single file needs to be processed, but more than one location for the file may need to be searched, you can usually find the file by specifying a file specification that contains a search list.

For example, consider the case of a directory that contains the files PAY.DAT and a backup copy of this file named PAY_BUP.DAT. You could specify a file name of PAY*.DAT in the file specification and invoke the Parse service once and the Search service once to locate either of the two files; this method will locate PAY.DAT before PAY_BUP.DAT. A potential problem arises if the file PAY.DAT has been deleted or renamed. In this case, unless the program determines that the file specification is one of several that are acceptable, any file whose name begins with PAY and has a file type of DAT could be accessed, for example, PAY_ACC.DAT.

You can avoid such problems by defining a search list logical name that specifies that VAX RMS search for PAY.DAT and then PAY_BUP.DAT. A search list named SEARCH could be defined as follows for the directory [SMITH]:

```
* DEFINE SEARCH [SMITH]PAY.DAT,[SMITH]PAY_BUP.DAT
```

To locate the file, you would specify SEARCH as the primary file. To locate the file, you would specify SEARCH as the primary file specification.

When the file locations to be searched reside in different directories, you could specify the ellipsis wildcard character in the directory field to search all subdirectories (if the file locations reside in the same directory tree). Alternatively, you could define a search list that searches for the file PAY.DAT in one directory, the same file name in a subdirectory, and then PAY_BUP.DAT in any directory in the directory tree by the following DEFINE command.

```
* DEFINE SEARCH [SMITH]PAY,[SMITH.PAY]PAY,[SMITH... ]PAY_BUP
```

You would use the file specification SEARCH:.DAT to locate the desired file. In this example, note that one of the search list file specifications contains wildcard characters; wildcard characters can be used in a search list just as with any other logical names and file specifications, if they are needed. However, the Parse and Search services must be used to locate the correct file.

When you need to locate files in different directory trees (or top-level directories), include complete directory specifications in your search list definition. You can also use search lists to locate files on different devices. For example, to locate the file TEST_DATA.DAT in the device/directory combinations of DISK1:[SMITH], DISK2:[STATS], or DISK2:[SMITH] you could define the search list TST as the following:

```
* DEFINE TST DISK1:[SMITH],DISK2:[STATS],DISK2:[SMITH]
```

To locate this file, you would specify TST:TEST_DATA.DAT.

Another case is to find the same directory and same file name on different devices; in this case, you might define TST as

```
⌘ DEFINE TST DISK1: ,DISK2: ,DISK3:
```

You could specify TST:[SMITH]TEST_DATA.DAT to locate the file. This example uses a single search list to locate files that would otherwise require multiple file specifications, even if wildcard characters were used.

5.3.2 Processing Many Files

To process many files using a single file specification, you always need to use the Parse and Search services to locate the files.

The application requirements and the directory location(s) of the file generally determine whether one or more search lists, wildcard characters, or search lists containing wildcard characters are used in the file specification. When files must be accessed in a certain nonalphabetical order, use a search list.

To process multiple files using a single file specification, invoke the Parse service (or its equivalent) once to interpret the file specification and create the file specification pattern to be searched. After the file specification is parsed, you can invoke the Search service to locate each file that matches the original file specification. In some cases, you can examine (or display) the resultant file specification string returned by the Search service to determine if you (or the terminal user) want to process (open) that file.

Should you want to list all file specifications that match a particular file specification and let the terminal user choose each file to be processed, wildcard characters can be used safely, possibly in a search list that contains wildcard characters in one or more of its file specifications. To reduce the number of files from which the user might choose the files to be processed, use a search list without wildcard characters or rely less on the wildcard characters. For example, to locate all files with a file type of DAT in a directory tree on different devices, you could define the search list TREE as follows:

```
⌘ DEFINE TREE DISK1: [MYDIR...],DISK2: [MYDIR...]
```

The primary file specification that would be used for the Parse service would be TREE:*.DAT. A great number of files might match this.

For applications that will need to locate certain files, search lists with more limited use of wildcard characters might be needed. Consider a file that contains a prefix of RESULTS followed by the date for which the data applies. You could use the file name RESULTS*JUN*.DAT to locate a record that was entered in the month of June by executing a Search service followed by an Open service for each file, reading all records until the correct one is found, and invoking the Close service after processing each file.

A search list should be used when a predefined group of files is processed by a program that is not intended to be interactive. Using a search list is particularly desirable if the files have unrelated file names or if they are located on different directories or devices. Another reason to use a search list is to minimize processing time by searching for a definite group of files.

5.3.3 Processing One or Many Files

For general-purpose applications, when the user enters a file specification that may indicate one file or many files, there is a means of testing whether one file or many files are to be processed, or to explicitly disallow the use of wildcard characters for applications where only a single file should be processed. To test for wildcard characters and/or search lists, invoke the Parse service and test the appropriate bits in the NAM\$L_FNB field.

The presence of a wildcard character usually indicates that many files should be processed, depending on program conventions. If a search list is present, it may or may not indicate that only one file should be processed and a convention is needed for users of that program. Thus, by testing whether a wildcard is present, the program can either invoke the Parse service once and the Search service repeatedly for each file to be opened or choose to disallow wildcard characters by requesting that the file specification be reentered. In some cases, the program may need to disallow the use of a search list or allow one or many files to be accessed, depending upon application conventions.

6

Advanced Use of File Specifications

This chapter is intended for readers who wish to more fully understand how VAX RMS internally applies defaults, parses file specifications, and handles directory specifications. This chapter also describes the use of rooted directory syntax and process-permanent files.

6.1 How VAX RMS Applies Defaults

This section gives details of how VAX RMS applies defaults in determining a file specified by your program. The three program-supplied file specifications are the primary file specification, the default file specification, and the related file specification. Of these, the primary file specification is usually specified. The default file specification might contain a default for the type component, such as .DAT to specify a data file, or it might supply a default for other file specification components. The related file specification is used when two files are involved in an operation, such as copying or merging files, in which the input file specification is the related file specification for the output file. The final default mechanism ensures that if the device and/or directory components are missing, certain process defaults will be used. The file specifications used in producing a full file specification that VAX RMS can use to locate a file are described in Table 6-1.

Table 6-1 File Specification Defaults

File Specification	Description
Primary	If the device field is a logical name, it will be repeatedly translated to its component parts. The resulting device name may be a physical device name, process-permanent file name, or concealed device logical name.
Default	If the device field is a logical name, it will be fully translated before defaults are applied. If any of the fields in the file specification from the previous step are missing, they will be replaced with the corresponding fields in the translated default file specification, where applicable.

Table 6-1 (Cont.) File Specification Defaults

File Specification	Description
Related	If the device field is a logical name, it will be fully translated and default values will be applied from the default file specification before defaults are applied from the related file specification). When used as an input file specification, missing component fields in the file specification (from the previous step) will be replaced by the corresponding fields in the related file specification. If fields contain wildcard characters (explained later), the wildcard characters remain in the fields. When used as an output file specification, the file name or file type fields will be replaced by the corresponding related file specification fields, where applicable. For more information including the use of multiple related file specifications, see Section 6.2.3.
Device and directory	If the device name is not present, the logical name SYSDISK will be used in the device field and, optionally, in the directory field. If SYSDISK cannot be translated to a physical device name, an error will occur. If the directory field is still missing, it will be replaced by the current process-default directory. The result of this step will be the fully defaulted and translated file specification that is actually used to locate the file.

The primary, default, and related file specifications can use logical names, which are first translated into their file specification components before that file specification is used to apply defaults or, in the case of the primary file specification, has defaults applied to it. If any components of the primary file specification are missing, the default file specification is used to apply defaults and the value for the missing component—if present in the default file specification—becomes part of the new primary file specification string. Similarly, if the file specification that is produced by applying the default file specification is still missing any components, the related file specification applies appropriate defaults, where applicable. If the primary file specification is still missing the device or directory name components, the process executing the program supplies default device and directory values.

Table 6-2 shows the sequence in which defaults are applied to unspecified components of a file specification and the string that results from each application. In Table 6-2, the program that specifies the primary file specification of FILE and a default file specification of the file type DAT. The process in which the program executes has a SYSDISK device of DISK1: and a process-default directory of [INV_C].

Table 6-2 Application of Defaults Example

String Name	String Applied	New String
Primary file specification	FILE	FILE
Default file specification	.DAT	FILE.DAT;
Related file specification	None.	FILE.DAT;
Default device (SYS\$DISK)	DISK1:	DISK1:FILE.DAT;
Default directory	[INV_C]	DISK1:[INV_C]FILE.DAT;
Resultant string		DISK1:[INV_C]FILE.DAT;1

The fully-parsed primary file specification is used to locate the file. The version number is appended to the fully defaulted and translated file specification (called the expanded file specification string) and that file specification becomes the resultant string.

When specifying the file specification information in a program, you can use the language keyword for the OPEN (or CREATE) statement; then specify the file specification characteristics using the FDL Editor and finally invoke the FDL\$CREATE (to create a file) or FDL\$PARSE and FDL\$RELEASE (to open a file) routines. Alternatively you can set the appropriate control block fields and call the VAX RMS services directly, perhaps as part of a USEROPEN or USER_ACTION routine.

Consider a program that does not explicitly specify the device and directory in any of the file specifications and does not have a related file specification. The process-default SYS\$DISK device and the current directory will be used in the fully defaulted file specification. However, if the data file needed is not in the current directory of the process running the program (or on the process SYS\$DISK device), the file will not be found. A good solution is to specify the data file's device and directory in the primary file specification, the default file specification, or the related file specification.

The program-supplied file specifications can be specified using the methods summarized below.

String	How You Can Specify It
Primary	Use the FDL attribute FILE NAME; use the file name or the name following the FILE, FILE_ID, or FILENAME keywords in the OPEN statement in certain VAX languages; or use the string pointed to by the FAB\$_FNA control block field.
Default	Use the FDL attribute FILE DEFAULT_NAME; use the default file specification or the name following the DEFAULTNAME or DEFAULT_FILE_ID keyword in the OPEN statement in certain VAX languages; or use the string pointed to by the FAB\$_DNA field.
Related	Use the name block that is pointed to by the NAM\$_RLF field; the related name block must specify the location of a file specification, which must be pointed to by the NAM\$_RSA field.

Consult the appropriate languages documentation for information about language statements and their keywords. Consult the *VAX/VMS File Definition Language Facility Reference Manual* for information about the FDL Editor, and the *VAX/VMS Utility Routines Reference Manual* for information about the FDL\$PARSE and FDL\$RELEASE routines. Consult the *VAX Record Management Services Reference Manual* for detailed information about VAX RMS control blocks and services.

Specifying all needed components in the primary file specification explicitly is a good practice because it decreases the chance of error. However, defaults are provided and can be very useful, especially for general-purpose applications and for applications in which the file specification is entered by the terminal user. Another option to consider is the use of logical names.

6.2 Understanding VAX RMS Parsing

In the following text, the term *expanded string* refers to the user-allocated string pointed to by the name block expanded string address (NAM\$_ESA) field; the term *equivalence string* refers to the area used internally by VAX/VMS that receives the result of a logical name translation.

As it processes each program-supplied file specification, VAX RMS translates the file specification into its component parts. Any component present in the primary, default or related file specifications are applied to the primary file specification to form the fully translated and defaulted primary file specification. This file specification string may be thought of as the pattern VAX RMS uses to locate the file. If a name block is present and the address and size of the expanded string has been specified, the full file specification is copied into the expanded string which is used to store the various intermediate forms of the file specification.

Note that the Parse service behaves differently than other services with regard to the expanded string. With the Parse service, the expanded string will contain any wildcard characters present in the file specification. The resultant string is not generated until another service is invoked, which typically uses the expanded string from the Parse service as input. When a search list is used, the expanded string contains the *first* location to be searched. VAX RMS keeps the information that specifies the remaining search list equivalence strings internally. As different file locations are examined, VAX RMS continually updates the expanded string to reflect the current location and the resultant string contains the actual file specification of the file that is found.

With the Create, Display, Erase, Open, and Search services, defaults are applied to the expanded string to select the actual file used. The resultant string can be used by the program to indicate which file was located. When the file is located, the version number found (or created) is appended to the resultant file specification string (not the expanded file specification string). When a search list is used, the resultant string contains the file specification where the file was actually found.

Although there are exceptions noted in the following sections that relate to advanced file specification use, VAX RMS produces a full file specification by going through the following steps as it examines the primary file specification.

6.2.1 Checking for Open-By-Name Block

If the open-by-name-block option is specified (FAB\$L_FOP field FAB\$V_NAM option is set), VAX RMS examines the name block for a valid device identification (NAM\$T_DVI field), directory identification (NAM\$W_DID field), and file identification (NAM\$W_FID field). If these fields are present, VAX RMS uses them to locate the file; all other components are ignored because they are not needed. If the open-by-name block succeeds, no expanded or resultant string is produced.

If these fields are not present in the name block or if an open-by-name block is not specified (for example, an Open service not preceded by a Parse service), VAX RMS performs the translation and application of defaults (see below). A file can also be created using the name block device and directory identification fields, but VAX RMS does not use the file identification.

If an open-by-name block is requested for remote DECnet file access between two VAX/VMS systems, VAX RMS does not check the device identification, directory identification or file identification to determine whether the requested open-by-name block operation can be performed. Instead, VAX RMS checks to see if a fully qualified resultant string is present.

If a fully qualified resultant string is not present, VAX RMS translates logical names and applies defaults as if an open-by-name block operation was not requested (see below).

6.2.2 File Specification Formats and Translating Logical Names

To form the fully defaulted and translated file specification, VAX RMS examines and attempts to translate each program-supplied file specification, beginning with the primary file specification string indicated by the contents of the FAB\$L_FNA and FAB\$B_FNS fields.

A file specification may have one of three formats:

- The first file specification is in the following format:

```
node::"foreign-file-spec"  
node::"task-spec-string"
```

VAX RMS attempts to translate the node name to determine if a logical node name is present; only a logical or physical node name (including an access control string, if present) is allowed if the translation is successful. If a logical node name is found, the translation is repeated. When translation cannot be performed, the file specification is copied directly into the expanded string. The quoted string is not parsed except to determine if it refers to a file or a task on the remote system. For additional information on the use of these formats, see the *VAX/VMS Networking Manual*.

- If the file specification contains only a name (without a terminating period or colon), VAX RMS attempts to translate it as a logical name. If the file name component field is translated successfully, the entire translation operation restarts, using the equivalence string as input. If the name is not translated successfully, it is used as the file name component and that file specification is ready for the application of defaults (for example, the file name FILE).
- If the file specification was not in either of the formats above, it is assumed to be in the full file specification format shown below.

```
node::device:[root.][directory]filename.type;version
```

Note that the use of brackets in the format do not imply optional elements. In fact, the node and the root are indeed optional elements and may not be present in a completed file specification.

VAX RMS isolates the various components, checks them for proper syntax, and copies them into the expanded string. If a node name is present, VAX RMS attempts to translate the node name as a logical node name as in step 1. If a name in the device component is present and

there is not a node name, VAX RMS attempts to translate the device name as a logical name. After repetitively translating a logical name, VAX RMS checks to determine whether the result of that translation contains a component already present for that file specification. If the primary file specification is being translated, VAX RMS signals an error if a duplicate component is detected. VAX RMS ignores (discards) the duplicate component created during the translation for the default and related string file specifications.

If a node name is not present and the name in the device component is not translated successfully, VAX RMS treats the name in the device component as a device name and the file specification is ready for the application of defaults.

If the logical name is translated successfully, VAX RMS performs one of the following actions:

- It checks the equivalence string to determine whether it refers to a process-permanent file. If a process-permanent file is being referenced, VAX RMS copies the logical name to the expanded string and terminates processing the file specification (defaults are not needed). The use of process-permanent files are discussed in Section 6.4.
- It checks the equivalence string to determine if the logical name is a concealed device logical name. If the logical name is concealed, and if no concealed logical names have been encountered previously in the device specification, the source string is used as the device name.
- If the equivalence string does not contain a process-permanent file and does not have the terminal attribute, VAX RMS restarts the translation operation using the equivalence string as input.

If a node name is present, VAX RMS passes the entire file specification (without the node name) to the remote node for interpretation, using the DECnet data access protocol (DAP) to communicate with the DECnet file access listener (FAL) at the remote node.

The logical name translation procedure reveals two conventions. First, if the file specification has been parsed previously by a VAX RMS file service, use the open-by-name-block option to save processing time. Second, a logical device name must be placed at the beginning of a file specification, unless it is preceded by a node name that indicates the node where the logical name should be translated.

6.2.3 Special Parsing Conventions

There are additional parsing conventions for advanced file specification features, such as a search list, related file specifications, and the way that directory specifications are handled.

Parsing Conventions for a Search List

VAX RMS uses several conventions when processing a search list logical name.

- When VAX RMS encounters a search list, it searches internally for the file using previously-specified search list file specifications. VAX RMS treats each file specification in the search list as a new file specification. That is, it does not use components of one file specification element in the search list as defaults for subsequent elements in the search list.

- With search lists, VAX RMS ignores the following errors:

Invalid device name (RMS\$_DEV)
Device not ready or not mounted (RMS\$_DNR)
Directory not found (RMS\$_DNF)
File not found (RMS\$_FNF)
Privilege violation (RMS\$_PRV)

All other errors terminate search list processing.

- When search list is embedded in another search list (nested), all file specifications of the nested search list are processed before the file specifications in the next-higher search list level. Therefore, VAX RMS permits iterative substitution in nested search lists as it does with other logical names. For example, consider two search lists X and Y:

```
# DEFINE X DISK1:[RED],DISK2:[WHITE]
# DEFINE Y X,DISK1:[BLUE]
```

When the search list Y is used, the search order for the device and directories is

- 1 DISK1:[RED]
- 2 DISK2:[WHITE]
- 3 DISK1:[BLUE]

- When opening a file, VAX RMS tries all search list possibilities before signaling an error completion status. If VAX RMS cannot find the file, it displays, where applicable, the final search list file specification together with the error completion message.

- When VAX RMS encounters a search list both the primary and secondary file specifications, it uses all of the search list elements in the primary file specification with the first element of the secondary file specification. Then it uses the all of the search list elements in the primary file specification with the second element of the secondary file specification. This process continues until VAX RMS tries every search list combination as it searches for the specified file.

For example, assume the program is looking for FILE.DAT which may be in one of two directories, [BIG] or [BEST], on one of two disks, DISK1: or DISK2:. First, the program must define two search lists, one for the disk (PRIM) and one for the directory (DEF):

```
* DEFINE PRIM DISK1,DISK2
* DEFINE DEF [BIG],[BEST]
```

Next, it must provide VAX RMS with a primary file specification that includes the search list (PRIM) for the disk together with the file name component:

```
PRIM:FILE
```

Finally, it must give VAX RMS the default specification that includes the search list (DEF) for the directory together with the file type component:

```
DEF:.DAT
```

Given this, VAX RMS looks for FILE.DAT using the file specification information in the following order:

Primary File Spec. (PRIM:TEST)	Default File Spec. (DEF:.DAT)	Expanded String
DISK1	[BIG]	DISK1:[BIG]TEST.DAT;
DISK2	[BIG]	DISK2:[BIG]TEST.DAT;
DISK1	[BEST]	DISK1:[BEST]TEST.DAT;
DISK2	[BEST]	DISK2:[BEST]TEST.DAT;

To carry this one step further, if this program provided a related file specification with a search list for FILE.DAT, VAX RMS would use all combinations of the search list elements in the primary and default file specifications with the *first* element of related file specification. Next, it would use all combinations of the search list elements in the primary and default file specifications with the *second* element of related file specification; and it would repeat this for each search list element in the related file specification.

- When a search list is used in a related file specification for an input file parse, special processing is needed to implement the DCL concept of temporary (sticky) defaults. The special processing needed to implement sticky defaults is the subject of the next section.

Special Processing for a Related File Specification

This section describes the special processing needed to implement sticky defaults when a search list is used in a related file specification for an input file parse.

The related file specification provides defaults when a related file name block is present. To use the related file specification, the file access block must specify the address of the primary file's name block (in the FAB\$_NAM field), and that name block must specify the address of the related file's name block (in the NAM\$_RLF field). The related file's name block must specify the address of a valid file specification in the resultant string (NAM\$_RSA and NAM\$_RSS) fields. Typically, a VAX RMS file service (other than Parse) places the file specification in the resultant string.

You can specify whether the related file will be used as an input file specification or an output file specification by setting (output file specification parse) or clearing (input file specification parse) the file-processing options (FAB\$_FOP) field output-file parse (FAB\$_OFF) bit option.

When an input file specification is being parsed, you can have multiple related name blocks by specifying the second related file's name block address in the NAM\$_RLF field of the first related name block, the address of the third related name block in the NAM\$_RLF field of the second name block, and so forth. The use of multiple related name blocks is especially useful for search lists; one related name block might contain a file type for use by any file specification in a search list, another might contain the full file specification that was produced by the first search list file specification, and another might contain the full file specification produced by the second search list file specification. This method allows the file specifications in a search list to provide sticky defaults, a characteristic associated with DCL command lines that contain multiple file specifications.

The term "sticky default" simply means that file specification components from the first file specification are applied as defaults to the next file specification component, eliminating the need, for instance, to specify the device specification for each file specification when all the files are located on the same device.

For a search list to be applied as a related file specification, the related file specification must not be an actual resultant string, but must include the search list logical name. The related file specification in this case must describe the original primary file specification. For example, consider the following search list definition:

```
# DEFINE WORK DISK1: [MINE] ,DISK2: [GROUP]
```

If your program wants to process lists of input files, such as WORK:A,B for example, your program must supply the string WORK:A as the related file specification, not DISK2:[GROUP]A.DAT. The routines LIB\$FIND_FILE and LIB\$FILE_SCAN are provided to perform this special processing; consult the *VAX/VMS Run-Time Library Routines Reference Manual* for additional information; also refer to Example 5-2, which shows how to call the LIB\$FIND_FILE routine.

Input File Specification Parse

When the file-processing options (FAB\$L_FOP) field output-file parse (FAB\$V_OFF) bit is clear and a node name is not present, VAX RMS processes the related file specification as an input file specification as shown below; the only wildcard character allowed is a single asterisk.

File Specification Component	Null Field Specification	Wildcard (*) Field Specification
Node	Use related file specification	Illegal
Device	Use related file specification	Illegal
Directory	Use related file specification	Remains wild
File name	Use related file specification	Remains wild
Type	Use related file specification	Remains wild
Version	Remains null	Remains wild

When the FAB\$L_FOP field FAB\$V_OFF bit is clear and a node name is present, VAX RMS processes the related file specification as an input file specification as shown below.

File Specification Component	Null Field Specification	Wildcard (*) Field Specification
Device	Remains null	Illegal
Directory	Remains null	Remains wild
File name	Use related file specification	Remains wild
Type	Use related file specification	Remains wild
Version	Remains null	Remains wild

Output File Specification Parse

When the FAB\$L_FOP field FAB\$V_OFF bit is set and a node name is not present, VAX RMS processes the related file specification as an output file specification as shown below.

File Specification Component	Null Field Specification	Wildcard (*) Field Specification
Node	Remains null	Illegal
Device	Remains null	Illegal
Directory	Remains null	Substitute from related file specification with restrictions
File name	Use related file specification	Substitute from related file specification
Type	Use related file specification	Substitute from related file specification
Version	Remains null	Substitute from related file specification

When the FAB\$L_FOP field FAB\$V_OFF bit is set and a node name is present, VAX RMS processes the related file specification as an output file specification as shown below.

File Specification Component	Null Field Specification	Wildcard (*) Field Specification
Device	Remains null	Illegal
Directory	Remains null	Substitute from related file specification with restrictions
File name	Use related file specification	Substitute from related file specification
Type	Use related file specification	Substitute from related file specification
Version	Remains null	Substitute from related file specification

As shown above, a wildcard character in an output directory specification is subject to certain syntax restrictions.

- Only the asterisk and the ellipses are permitted in the output directory specification. These two wildcard characters cannot both be present in the same related file specification output directory specification except as the directory specification [...].
- A subdirectory specification that contains wildcard characters cannot be followed by a subdirectory specification that does not contain wildcard characters.
- Specifications in the UIC directory format may receive defaults only from directories in the UIC directory format.
- Specifications in the non-UIC directory format may receive defaults only from directories in the non-UIC directory format.
- Specifications in the non-UIC directory format that consist entirely of wildcard characters may receive related file specification defaults from directories in UIC or non-UIC format.

VAX RMS processes wildcard characters in an output directory specification as follows:

- If you specify an output directory using a specification that consists entirely of wildcard characters ([*] and [...] only are allowed), VAX RMS will substitute the entire directory from the related file specification. This substitution allows you to duplicate an entire directory specification.
- If you specify an output directory with a trailing asterisk (for example, ([A.B.*])), VAX RMS will substitute the first "wild" subdirectory from the related file specification. This substitution permits you to move files from one directory tree to another directory tree that is not as deep as the first one.

- If you specify an output directory with a trailing ellipses (for example, ([A.B...])), VAX RMS will substitute the entire "wild" subdirectory from the related file specification. This substitution permits you to move entire subdirectory trees.
- The related name block must have the appropriate file name status bits set in the file name status bits (NAM\$L_FNB) field set according to the resultant string to allow VAX RMS to identify the "wild" portion of the resultant string.

6.3 Directory Syntax Conventions and Directory Concatenation

One of the components of a file specification is the directory specification. VAX RMS supports two conventions or types of directory specifications, one of which is used more often than the other.

When VAX RMS applies defaults to a directory specification in a file specification, the rules differ depending on what type of a directory specification is present. There are two directory syntax conventions available to access directories: normal and rooted. The default directory access is normal syntax. That is, you can specify the directory desired using the directory syntax described in the *VAX/VMS DCL Dictionary*.

6.3.1 Using Normal Directory Syntax

There is a master file directory (MFD) on each disk volume. Within each MFD, top-level directories are cataloged using the DCL command CREATE /DIRECTORY (or equivalent services). Beneath each top-level directory, you can create subdirectories referenced from the top-level directory.

Once the subdirectories are created, you can create subdirectories referenced from the each subdirectory. You can create a maximum of seven levels of subdirectories beneath a top-level directory. Collectively, the subdirectories below a directory and the subdirectories within the subdirectories (and so forth) are referred to as a *directory tree*. A physical structure that resembles a directory tree would be a pyramid or the roots of a tree.

The base point for normal directory syntax access can be relative to the current position in the directory tree or an absolute reference that explicitly or by default states any higher-level directories or subdirectories needed to identify the appropriate directory or subdirectory. An absolute directory reference begins with a directory name; a relative directory reference begins with a hyphen (-) or period (.) as the initial part of a directory specification. An absolute reference might include the name of the top-level directory and one or more subdirectories. A relative directory reference relies on the use of the process-default directory and device, which are set using the

DCL command SET DEFAULT. Refer to the *VAX/VMS DCL Dictionary* for additional information and examples.

A relative directory reference can be in one of several forms. Assume the current directory position (process-default directory) is [SMITH.JONES].

- You can specify a lower level in the directory tree with a period (.) to indicate that the current directory position ([SMITH.JONES]) is prefixed to the specified directory. For example, when you specify the following directory:

[.DATA]

This directory specification is combined with the current directory position to form [SMITH.JONES.DATA].

- You can specify higher or parallel levels in the directory tree by beginning the directory specification with a hyphen (-) to indicate that the current directory position ([SMITH.JONES]) is prefixed to the specified directory after ascending one directory level for each hyphen specified. For example:

[-]

This directory specification refers to the directory [SMITH], because a single hyphen requests a directory one level higher than the current directory position.

The following directory specification refers to [SMITH.BROWN]:

[-.BROWN]

In this case, a subdirectory of [SMITH] is accessed, [SMITH.BROWN], which is at the same (parallel) level in the directory tree as the current default position. The hyphen (ascend) and the period (descend) combine to produce this traversal of the directory tree. Two successive hyphens can also be used:

[--.BLACK.DATA]

In this case, the directory location ascends two directory levels and then down two levels to the directory [BLACK.DATA] from the current directory position, [SMITH.JONES].

- You can refer to the current directory position explicitly by specifying an empty directory specification:

[]

This refers to the current directory position, [SMITH.JONES].

A directory name that is the initial portion of a directory specification is always interpreted as a top-level directory, or an absolute directory reference. For example, the following directory specification refers to a top-level directory, [GREEN], regardless of the current default directory:

[GREEN]

Conversely, a period or a hyphen before a directory name is always associated with a relative directory reference.

If the program omits either the device or the directory element, VAX RMS translates SYS\$DISK. Therefore, any directory fields yielded by translation of SYS\$DISK override the process default directory. If translation of SYS\$DISK does not yield the directory element, VAX RMS uses the process default directory. Note that you can change the process default directory using the SET DEFAULT command or the SYS\$SETDDIR service.

6.3.2 Rooted Directory Syntax Applications

The rooted directory syntax convention for accessing directories is made possible through the use of concealed device logical names. The use of rooted directory syntax allows programs (or procedures) to refer to directory trees as logical devices and to top-level directories. A reference to a top-level directory actually accesses existing subdirectories without program modification; thus, rooted directory syntax extends the flexibility associated with logical names. Similarly, the use of rooted directory syntax can reduce the number of top-level directories needed for a disk volume. Another important use of rooted directory syntax is to allow multiple VAX/VMS system directory trees for a single system volume.

You specify rooted directory syntax by using a certain type of logical name either in a program-specified file specification or in the device/directory for a SET DEFAULT command. If a program specifies a logical device name in the file specification, the logical name can be redefined to specify a rooted directory logical name, which changes the directory (and the file or files) accessed by the program without program modification.

If a program does not specify a logical device name in the file specification, the user (or a command procedure) can issue DEFINE commands and the SET DEFAULT command before running the program to indicate the use of rooted directory syntax and to specify the process-default device/directory. Such use of the SET DEFAULT command changes the directory accessed by the program without requiring that you modify the program. After the program completes, use the SET DEFAULT command again to specify the new process-default device/directory and resume the use of normal directory syntax (if desired).

By eliminating the need to modify the program, the program does not have to be recompiled (or reassembled), relinked, and fully retested. Also, the programs do not have to be recopied to the appropriate locations.

6.3.3 Using Rooted Directory Syntax

Rooted directory syntax is a means of specifying a directory (or subdirectory) that will appear to the user's process to be, not the specified directory, but the master file directory (MFD) for the logical disk volume. Subdirectories of the rooted directory appear as top-level directories (user file directories) for the logical disk volume.

The directory specified during logical name definition serves as a base from which directory access down the directory tree can occur and is referred to as the *root directory*. Root directories must be specified using alphanumeric UICs; octal numbering for group and member designations is not allowed.

A concealed device logical name that defines not only a hidden device name but also a hidden root directory is called a *rooted-device logical name*. With VAX/VMS Version 4.4, VAX RMS no longer assumes that a concealed logical name is a terminal logical name.

When you define the rooted-device logical name for subsequent use in a program or in a SET DEFAULT command, you specify that the logical name is a concealed device logical name by using the /TRANSLATION_ATTRIBUTES=CONCEALED qualifier for the DCL command DEFINE or ASSIGN. To define the concealed device logical name as a rooted-device logical name, the root directory must contain a trailing period (.), such as DUA22:[ROOT.]. The directory specification of a trailing period is only allowed for the root directory.

Once a root directory has been defined, all subsequent directory references will refer to the specified root directory or any of the directories in the directory tree below the root directory. As part of the rooted directory syntax conventions, a normal directory reference to a top-level directory instead refers to a subdirectory of the specified root directory when using rooted directory syntax. This is consistent with the fact that the root directory appears as the MFD; a reference to the directory [000000] when using rooted directory syntax refers to the root directory. Similarly, the root directory is shown as directory [000000] as output to SHOW DEFAULT and other commands. Note that the directory specification [0,0] is not a valid equivalent to [000000] when using rooted directory syntax. The actual name of the rooted directory is not accessible.

For example, assume the top-level directory [ROOT1] on disk DUA7 contains a subdirectory [ROOT1.SUB]. The directory [ROOT1] is defined as the root directory associated with the logical name BASE as follows:

```
DUA7: [ROOT1.]
```

You can refer to [ROOT1.SUB] using the syntax BASE:[SUB]. The following summary of the actual directory accessed and the equivalent rooted directory syntax applies to this example.

Actual Directory	Rooted Syntax	Comments
DUA7:[ROOT1]	BASE:[000000]	[ROOT1] appears as the MFD
DUA7:[ROOT1.SUB]	BASE:[SUB]	[ROOT1.SUB] appears as a top-level directory

The following example shows how to actually define the root logical name described above and how to access a subdirectory of the specified root directory. Note that the trailing period at the end of the directory name [ROOT1.] is what indicates that a root directory is present.

```
# DEFINE /TRANSLATION_ATTR=CONCEALED BASE DUA7: [ROOT1.]
# SET DEFAULT BASE: [SUB]
# DIRECTORY *.DIR, [-]*.DIR
```

```
BASE: [SUB]
SUBSUB.DIR
BASE: [000000]
SUB.DIR
```

In this example, the SET DEFAULT command defines the process-default directory as [ROOT1.SUB] using the rooted-device logical name BASE. The following DIRECTORY command looks for directory files (DIR file type) within the current directory ([ROOT1.SUB]) and then in the next highest directory ([ROOT1]). The directory [ROOT1.SUB] is listed (by the DIRECTORY command) as a top-level directory (BASE:[SUB]) and the root directory is listed using the syntax of the MFD, BASE:[000000]. The logical name BASE is displayed because it is concealed.

6.3.4 Concatenating Rooted Directory Specifications

The syntax rules that VAX RMS uses when concatenating directory specifications for rooted directory syntax differ from the rules VAX RMS uses when using normal directory syntax. One difference between the two conventions is associated with the trailing period in the root directory definition. For example, consider how a top-level directory reference is handled. With rooted directory syntax, the root directory's trailing period is implied as a leading period in subsequent rooted directory references.

Directory concatenation *within the same file specification* occurs only using a rooted-device logical name. Normal directory concatenation occurs only through the application of defaults. Rooted directory concatenation can occur within the same file specification and through the application of defaults. Rooted-device logical names specify a device name and a root directory. VAX RMS translates a rooted-device logical name into the device and root directory before it merges any other parts of a file specification (if present) into the equivalence file specification.

When a rooted-device logical name is used in combination with a directory specification, the following rules apply:

- You can refer to the root directory itself. The syntax of [000000] and certain relative directory references refer to the root directory when using a rooted directory syntax.

You can never refer to a directory *above* the specified root directory when using rooted directory syntax. Thus, the root directory appears like the MFD whenever a subsequent directory specification is used. For example, when your process-default directory is the root directory, a reference to [-] results in an error. For example:

```
# DEFINE /TRANSLATION=CONCEALED BASE DUA7:[ROOT1.]
# SET DEFAULT BASE:[000000]
# DIRECTORY *.DIR
BASE:[000000]
No files found
# DIRECTORY [-]*.DIR
%DIRECT-E-OPENING, error opening [-]*.DIR as input
-RMS-E-DIR, error in directory name
```

- You can refer to a specific subdirectory of the root directory in the same manner you would refer to a top-level directory using normal directory syntax. For example:

```
# DEFINE /TRANSLATION_ATTR=CONCEALED BASE DUA7:[ROOT1.]
# SET DEFAULT BASE:[SUBDIR]
```

- You can refer to any subdirectory in the directory tree below the root directory. Wildcard characters are supported for traversing directory trees. You can refer to all directories below the root directory [...], all directories one level below the root directory [*], all directories two levels below the root directory [*.], and other combinations. For example:

```
$ DEFINE /TRANSLATION_ATTR=CONCEALED BASE DUA7:[ROOT1.]
$ DIR BASE:[*...]*.DIR

BASE:[SUBDIR]
    SUBSUBDIR.DIR
BASE:[SUBDIR.SUBSUBDIR]
    SUBSUBSUBDIR.DIR
BASE:[OTHERSUB]
    OTHERSUBSUB.DIR
```

Another difference between the conventions VAX RMS uses for rooted directory syntax and standard directory syntax is the number of permissible nested directory levels.

Prior to VAX/VMS Version 4.0, VAX RMS restricted the use of nested directories to a maximum of eight levels. With rooted directory logical names, however, you can "hide" eight additional levels in the rooted directory logical name and effectively nest 16 levels of directories.

You must access these files using the rooted directory logical name. Whenever VAX RMS tries to create a file name with more than eight normal directory levels, it returns an error. For example, if you define the rooted directory logical name MYROOT to be DUA0:[D1.D2.D3.D4.D5.D6.] and then nest six more subdirectories, you have created the following legal file specification:

```
MYROOT:[D7.D8.D9.D10.D11.D12]name.type
```

If you then try to access this file as:

```
DUA0:[D1.D2.D3.D4.D5.D6.D7.D8.D9.D10.D11.D12]name.type
```

VAX RMS returns a status code indicating the file specification is illegal. The problem can occur in a number of situations. For example, if you try to back up your directory tree DMA0:[D1...], the Backup Utility only backs up the first eight levels of the directory tree.

Note

This problem does not occur when you back up an entire disk at one time, that is, if you are using the BACK/IMAGE command or the BACKUP/PHYSICAL command.

The process-default device and directory are used *indirectly* as defaults. The expanded rooted-device logical name device and root directory are used before the process-default device and directory. You can use relative directory syntax, such as the hyphen (-) and leading period (.name). When a directory component is missing, VAX RMS attempts to use the process-default directory.

Consider the rooted-device logical name X that is defined as

```
# DEFINE/TRANSLATION_ATTR=CONCEALED X DJB3:[SMITH.]
```

When the rooted-device logical name X is used in combination with a directory specification, all directory references are relative to the root directory [SMITH.]. Most wildcard characters that apply to normal directory syntax also apply to rooted directory syntax. The following combinations of the rooted-device logical name X and various directory names, coupled with the current process-default directory of DJB3:[JONES], access the directories listed below.

File Spec.	Actual Directories Accessed
X:	[SMITH.JONES]
X:[000000]	Root directory, [SMITH]
X:[]	[SMITH.JONES]
X:[]	Root directory [SMITH], listed as X:[000000]
X:[]	Illegal (error)
X:[name]	[SMITH.name]
X:[.name]	[SMITH.JONES.name]
X:[name.*...]	All directories in all directory trees below [SMITH.name]
X:[*]	All directories one level below [SMITH]
X:[*...]	All directories in all directory trees below [SMITH]
X:[...]	All directories in all directory trees below [SMITH.JONES]

Note that the process-default directory is used when the directory specified is a relative directory reference; this is because the specified directory name contains a leading period or a hyphen.

6.3.5 An Example of Rooted Directory Use

Consider an application consisting of several programs that reference the same file using a file specification `IN:[INVENTORY]FILE.DAT`. Previously, the logical name `IN` was defined as `DUA29`.

This example assumes the programs were previously invoked using a command procedure that defined the logical name `IN` to be `DUA29`, as shown below:

```
# ON CONTROL_Y THEN GOTO ENDIT
# DEFINE IN DUA29:
# RUN XYZPROG
# ENDIT:
# EXIT
```

The programs have been dispersed to many users within the company to show the current inventory level and stockroom for a particular item. Over time, however, as the company grew, the number of parts in the inventory grew and the number of inventory records grew also, making the indexed sequential file used extremely large, which increased access time. Because there was a natural breakdown of the types of parts into about four groups, the file was split up. A special-purpose program was designed to read each record in the large file, determine the type of record, and write the record into the appropriate file on the appropriate device. The output files were each defined as `FILE.DAT`, but in a top-level directory associated with their category. For example, computer supplies files were cataloged in the directory `[COMPUTER.INVENTORY]`.

This command procedure was modified to request the type of part needed, which conditionally defines the value of `IN` to be a rooted-device logical name with the appropriate top-level root directory. For example, if the terminal user requested a computer part, the following DCL command was executed within the command procedure:

```
# DEFINE/TRANSLATION_ATTR=CONCEALED IN DUA29:[COMPUTER.]
```

Thus, the program accessed `IN:[INVENTORY]FILE.DAT` as it did before, but the logical name caused it to reference `DUA29:[COMPUTER.INVENTORY]FILE.DAT` and not `DUA29:[INVENTORY]`. The set of programs never had to be modified, recompiled (or reassembled), relinked, and copied; only the command procedure is enhanced and is shown below in Example 6-1.

Example 6-1 Example of Rooted Directory Syntax

```

* ON CONTROL_Y THEN GOTO END
*   GOTO ASK
* RETRY:
*   WRITE SYS$OUTPUT "Enter a number from 1 to 4 for the type of part"
* ASK:
*   WRITE SYS$OUTPUT -
*     "Select Parts Group: 1-COMPUTER 2-TYPEWRITER 3-DESK 4-OTHER 5-END"
*   INQUIRE ANS
*   IF ANS .GT. 5 .OR. ANS .LT. 1 THEN GOTO RETRY
*   IF ANS .EQ. 5 THEN EXIT
*   IF ANS .EQ. 1 THEN DEFINE/TRANS=CONCEAL IN DUA29:[COMPUTER.]
*   IF ANS .EQ. 2 THEN DEFINE/TRANS=CONCEAL IN DUA29:[TYPEWRITER.]
*   IF ANS .EQ. 3 THEN DEFINE/TRANS=CONCEAL IN DUA29:[DESK.]
*   IF ANS .EQ. 4 THEN DEFINE/TRANS=CONCEAL IN DUA29:[OTHER.]
*   RUN XYZPROG
* END:
*   EXIT

```

6.4 Using Process-Permanent Files

A process-permanent file is one that is opened or created through VAX RMS by supervisor or executive mode code when the process-permanent (FAB\$V_PPF) bit is set in the file-processing options (FAB\$L_FOP) field. This causes the VAX RMS-maintained internal data structures to be allocated in an area of memory in the process control region that remains allocated for the life of the process. Thus, process-permanent files can remain open across image activations. SYS\$COMMAND, SYS\$INPUT, SYS\$OUTPUT, and SYS\$ERROR are all opened in this manner by the LOGINOUT command image.

Files opened for processing through the use of the DCL command OPEN are also opened in this manner. You cannot directly access process-permanent files by user mode code; you can, however, access them indirectly. VAX RMS provides a subset of the total available operations to the indirect accessor.

Indirect accessors gain access to process-permanent files through the logical name mechanism, as follows:

- 1 The LOGINOUT command image, or at a later point the command interpreter, opens or creates a file corresponding to the process's command, input, output, and error message streams. Logical names are created in the process logical name table for SYS\$COMMAND, SYS\$INPUT, SYS\$OUTPUT, and SYS\$ERROR, respectively. The equivalence string for the logical name has a special format that indicates

the correspondence between the logical name and the related process-permanent file. For more detail concerning the equivalence-string format for logical names, see the discussion of logical name services in the *VAX/VMS System Services Reference Manual*. For example, for an interactive user, these single process-permanent files are opened for the terminal.

- 2 When an indirect accessor opens or creates a file specifying a logical name that has one of these special equivalence strings, VAX RMS recognizes this and therefore does not open or create a new file. Instead, the returned value for the internal file identifier (and later the value for the internal stream identifier from a Connect service) is set to indicate that access to the associated process-permanent file is with the indirect subset of allowable functions.

The implications for the indirect accessor are listed below:

- A Create service for a process-permanent file becomes an Open service; the fields of the FAB are output according to the description of the Open, not the Create service.
- The Open or Create service requires no I/O operations.
- Any number of indirect Open and Create operations are allowed.
- There is only one position context for the file; each sequence of the Open or Create service accesses the same record stream, not an independent stream.
- If the process-permanent file was initially opened with the sequential-processing-only (FAB\$V_SQO) bit set in the FAB\$L_FOP field, neither random access nor the Rewind service is permitted. This is the case for SYS\$COMMAND, SYS\$INPUT, SYS\$OUTPUT, and SYS\$ERROR.
- Certain options to various services produce errors. For example, you cannot set the non-file-structured (FAB\$V_NFS), process-permanent file (FAB\$V_PPF), and user-file-open (FAB\$V_UFO) bits of the FAB\$L_FOP field for the Open and Create services. Other options are ignored, such as the spool (FAB\$V_SPL), submit-command-file (FAB\$V_SCF), and delete (FAB\$V_DLT) bits of the FAB\$L_FOP field for the Close service, the asynchronous (RAB\$V_ASY) bit of the RAB\$L_ROP field, and both the multiblock count and multibuffer count fields (RAB\$B_MBC and RAB\$B_MBF).
- If a name block is used and either an expanded or resultant file specification string is returned, the string consists solely of the process logical name followed by a colon (such as SYS\$INPUT:).

- The file access (FAB\$B_FAC) field is ignored on an Open service; instead, operations are checked against the FAB\$B_FAC field specified for the original Open or Create service.
- Information from the record attributes field is saved on each Open service and subsequent Connect service in the values returned in the internal file identifier and internal stream identifier fields. If the output file is a print file (variable with fixed-control record format and the FAB\$V_PRN bit is set in the record attributes field), mapping is performed for each Put service from the user-specified carriage control to the print file carriage control format. Thus, different carriage control types from different indirect Open Services all work correctly.
- You cannot use the Erase service.
- Checking is performed for \$DECK, \$EOD, and other dollar-sign (\$) records on the SYS\$INPUT stream if the SYS\$INPUT stream is from a file. Checking is not done if SYS\$INPUT comes from a record-oriented device, such as a terminal or mailbox. (see the *VAX/VMS DCL Dictionary*).
- At image exit time, the VAX RMS run-down control routine ensures that the indirect I/O on process-permanent files terminates; however, these files are not closed.
- All file organizations may be opened directly as process-permanent files (for example, through the DCL command OPEN), but only those files with a sequential organization may be indirectly accessed.



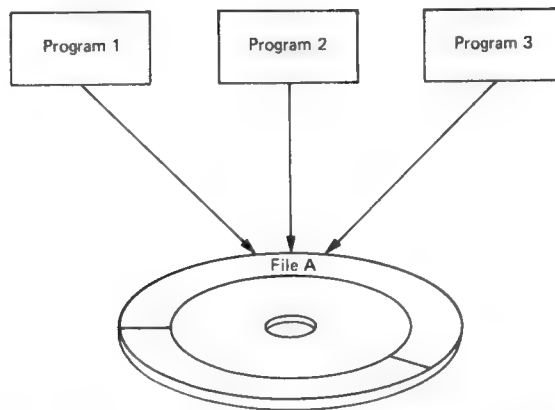
7

File Sharing and Buffering

This chapter discusses the run-time options that are available when opening, connecting and closing a shared file. These options are also implicit in creating a shared file because the Create service includes an implied file open, which uses the run-time options either explicitly or by default.

File sharing includes file accessing, record locking, and local and shared buffering. Figure 7-1 shows a typical shared file situation.

Figure 7-1 Shared File Access



ZK-757-82

7.1 File Sharing

VAX RMS file sharing allows multiple users to access a single file. Timely access to files sometimes requires that more than one active program be allowed to read, write, and modify records within the same file simultaneously. Whether or not a file can be shared depends on the type of device it resides on and the explicit file-sharing information specified by the accessing programs.

Magnetic tape files cannot be shared because magnetic tape drives are sequentially operated devices. However, disk files can be shared by any combination of readers and writers without restriction. Your program provides the information that enables file sharing. You control the degree of sharing by providing VAX RMS with an explicit file-sharing specification when your program opens or creates a file. This specification indicates the types of file operations that are permitted for user programs that share the file.

When a program creates or opens a disk file, it gives VAX RMS two pieces of information needed to determine if and how the file may be shared. First, it states the types of operations it intends to perform on the file, such as read, write, or update. VAX RMS subsequently checks this information to protect against unauthorized file access.

Second, the program must specify the types of operations other concurrently active programs can perform on the file. When the sharing specification of one program is compatible with the sharing specification of another, both programs can gain access to the file simultaneously. To ensure that multiple programs can access the file simultaneously, you may have to do some schedule planning.

7.1.1 Types of File Sharing and Record Streams

A single user process can access the same file using different methods of record access, or *record streams*. Each record stream is associated with a RAB that defines and maintains the information used to access records by way of the record stream.

You should consider whether the file will be maintained on line similar to a real-time application or as a master file updated periodically from one or more transaction files. The latter method can substantially improve performance when many users share a master file. If users are only reading the file, you can eliminate record locking to improve performance, especially for applications where many users access the file and for files that reside on a shared VAXcluster volume. Where system resources permit, you may consider the use of global buffers for read-only applications.

When an application requires current online information, record modifications and additions should be made as quickly as possible to minimize the time that other users are unable to access the record being updated and to minimize the additional overhead of checking record locks. For instance, when records are updated interactively by the terminal user, it is advantageous to minimize the amount of time the record remains locked by allowing other users to read, but not write, that record. Reducing the time

spent for record modifications or additions is especially important when the file resides on a VAXcluster-shared disk volume (see Section 3.6 for more information.)

Two options that are especially important for shared files are the file-access and file-sharing options. These options specify the type of record access that the sharing processes can perform. They are specified by the FDL attributes ACCESS and SHARING and the VAX RMS FAB fields identified by the symbolic offsets FAB\$B_FAC and FAB\$B_SHR. When creating or opening a file, VAX RMS compares the values of these fields to determine whether or not the requesting process may have access to the file.

The first process to access a file determines what operations other users can perform on the file, which in practice, determines whether or not subsequent users are allowed to access the file. For example, if your process requests a certain type of access that the first user to access the file (since the file was closed last) has disallowed, your process will not be granted access.

When choosing the access other users may have to the file, you can specify the type of file sharing to be used, and indicate whether or not other record streams can access the file simultaneously. In a VAXcluster environment, processes can access shared files on the same or different nodes of the same VAXcluster (see Section 3.6).

Each record stream is represented by a RAB. Independent record streams are connected to a single file in one of three ways:

- Within a single process or across several processes, multiple FABs can be connected to a shared file. One or more record streams may then be connected to each FAB. This form of sharing is known as *interlocked interprocess file sharing* and is associated with reading or writing records, not blocks.
- Within the same process, multiple record streams can be associated with a single FAB to read and write records, not blocks. This form of sharing is known as *multistreaming*.
- Within a single process or across several processes, multiple FABs can be connected to a file. One record stream (RAB) is connected to each FAB and the user processes provide their own synchronization outside of VAX RMS. This form of file sharing is known as *user-interlocked interprocess file sharing*. (It usually applies only to block I/O processing and to record processing for non-shared sequential files residing on disk devices.)

A single file can be accessed by both interlocked interprocess file sharing and multistreaming. DIGITAL does not recommend the simultaneous use of interlocked interprocess file sharing and user-interlocked interprocess file sharing on the same file if the user who requests user-interlocked interprocess

file sharing intends to modify the file. The reason is that record locking will not be done or checked for the processes using user-interlocked interprocess file sharing.

You must define the type of access *your process* will have based on the types of record operations it will perform. The record operations with associated FDL and VAX RMS options are summarized in Table 7-1.

Table 7-1 File Access Record Operations

Function (VAX RMS service)	FDL and VAX RMS Options
Read records (Get)	ACCESS GET specified or FAB\$_FAC field FAB\$_GET set
Locate records (Find)	ACCESS GET specified or FAB\$_FAC field FAB\$_GET set
Delete records (Delete)	ACCESS DELETE specified or FAB\$_FAC field FAB\$_DEL set
Add new records (Put)	ACCESS PUT specified or FAB\$_FAC field FAB\$_PUT set
Truncate file (Truncate)	ACCESS TRUNCATE specified or FAB\$_FAC field FAB\$_TRN set
Modify records (Update)	ACCESS UPDATE specified or FAB\$_FAC field FAB\$_UPD set
Access blocks (see text)	ACCESS BLOCK_IO specified or FAB\$_FAC field FAB\$_BIO set; under certain conditions, ACCESS RECORD_IO or FAB\$_FAC FAB\$_BRO

The record-access functions you request are compared with the specified file's protection. If your process is limited to reading and locating records, it must have read access to the file. If your process will be deleting, adding, truncating, or updating records, it must have write access to the file. VAX RMS permits any process that may delete, add, truncate, or modify records to also locate and read records because write access to a file also implies read access.

To access blocks using VAX RMS, your process can use the Read, Space, and Write services, not normal record I/O; this option is usually only used by applications written in VAX MACRO or other low-level languages. Note that when ACCESS BLOCK_IO is specified, the user must also specify either SHARING USER_INTERLOCK or SHARING PROHIBIT.

Different types of record operations can be specified to define the type of access to be allowed for other processes, as shown in Table 7-2.

Table 7-2 File Sharing Record Operations

Function (VAX RMS Service)	FDL and VAX RMS Options
Read records (Get)	SHARING GET specified or FAB\$B_SHR field FAB\$V_SHRGET set
Locate records (Find)	SHARING GET specified or FAB\$B_SHR field FAB\$V_SHRGET set
Delete records (Delete)	SHARING DELETE specified or FAB\$B_SHR field FAB\$V_SHRDEL set
Add new records (Put)	SHARING PUT specified or FAB\$B_SHR field FAB\$V_SHRPUT set
Modify records (Update)	SHARING UPDATE specified or FAB\$B_FAC field FAB\$V_SHRUPD set
No access	SHARING PROHIBIT or FAB\$B_SHR field FAB\$V_NIL set
User interlocking	SHARING USER_INTERLOCK or FAB\$B_SHR field FAB\$V_UPI set
Multistreaming	SHARING MULTISTREAM or FAB\$B_SHR field FAB\$V_MSE set

If other processes are limited to reading and locating records, they will be unable to modify or add records, and record-lock checking will not be performed. If other processes are allowed to delete, add, or modify records, they will also be able to read records; however, record-lock checking will occur. All record-access functions use interlocked interprocess file sharing.

No access denies access to all users except the user who specifies this option. The no-access option might be used when a file is shared infrequently or to perform a major file update. When using this option, close the file promptly if other users may need to access the file. Choose this option or the user-interlocking option when using block access; to use the Queue I/O Request system service, specify the FILE USER_FILE_OPEN attribute (FAB\$L_FOP field FAB\$V_UFO set). The no-access option does not allow file sharing and requires that your process have write file protection access.

User interlocking permits the user, not VAX RMS, to maintain interlocking protection (including maintaining the end-of-file mark). For any other form of file sharing, VAX RMS controls the reading and writing of I/O buffers to ensure the integrity of file and record structures. This option is useful for non-shared sequential files and for block I/O access using VAX RMS or the Queue I/O Request system service.

Multistreaming allows your process to access the same file using more than one record stream and allows other users to access the file using interlocked interprocess file sharing (unless SHARING PROHIBIT is also specified). When specifying this option, the appropriate SHARING record operations should also be specified, such as SHARING GET. When multiple streams are connected, the buffers allocated for each stream become part of a buffer cache for the entire process. (A buffer cache is a common shared buffer pool intended to minimize I/O.) A record operation on one stream may use cached buffers from a previous record operation on a different stream that referenced the same buckets.

Your program must specify both the file access and file sharing desired when the file is opened or created. Certain combinations are allowed and are likely to be used; others return errors. For example, to read records from the file and allow others to read records (ACCESS GET and SHARING GET) is allowed and is the default when using FDL or VAX RMS and some languages to open a file, depending on the language used. To read records and allow others to write records is also allowed, but is not usually the default (for example, ACCESS GET with SHARING PUT and SHARING UPDATE). You can write records while allowing others to write to the file (for example, ACCESS UPDATE with SHARING PUT and SHARING DELETE).

When creating a file using the Create service, the default for FDL, VAX RMS, and certain languages is to write records to the file and not allow others to access the file (ACCESS PUT and SHARING NONE). When creating a file with the create-if option, it is especially important to specify the access and sharing values. For example, when using Create service with the create-if option and the file already exists, if another user has already opened the file you will be denied access to the file because you implicitly requested to be the only user accessing the file by specifying SHARING NONE or its equivalent. One way to override this default is to allow most or all operations for other users (such as SHARING GET, SHARING PUT, SHARING UPDATE, and SHARING DELETE).

Combinations of file access and file sharing that specify a mixture of interlocked interprocess file access and user-interlocked interprocess file sharing allow the user to access the file without record-locking protection. Such combinations are not recommended for general use; they should be used only for applications that require read-only access to a file. Other combinations may cause an error, such as requesting ACCESS BLOCK_IO without specifying SHARING NONE or SHARING USER_INTERLOCK.

7.1.2 Interlocked Interprocess File Sharing

The interlocked interprocess form of file sharing is used most frequently. This method allows the connection of one or more record streams (RABs) to one or more processes (FABs), either within a single process or across several processes. When using this form of file sharing, the combination of values specified for file sharing and file access specified by the initial accessor of the file determines the type of file access that will be allowed for subsequent users.

The initial accessor must be aware of the restrictions that result from the combination of values specified by file access and file sharing. A typical example would be when the initial accessor denies any kind of write access to subsequent users. Such a restriction can occur when the initial accessor specifies some type of write access for file access without specifying write access for file sharing.

If the initial accessor specified read-only file access and file sharing, then subsequent users will be able only to read the file. If the appropriate type of write access is not specified, then subsequent users will not be able to perform the corresponding write operations to the file.

If the initial accessor specifies one or more values for file sharing, subsequent users can access the file, provided that they specify a matching value for file access. Thus, if the initial accessor specifies SHARING GET and SHARING PUT, then subsequent users must specify ACCESS GET and ACCESS PUT to have read and write access to the file, or only ACCESS GET to read the file or ACCESS PUT to read and write to the file (read access is implied by write access).

Table 7-3 presents the values that the initial accessor of a file can specify for file sharing to permit access to subsequent users.

Table 7-3 Initial File Sharing and Subsequent File Access

Initial Accessor Sharing	Subsequent Accessor Access
SHARING PROHIBIT	No access allowed
SHARING GET ¹	ACCESS GET ¹
SHARING DELETE	ACCESS DELETE
SHARING PUT	ACCESS PUT
SHARING UPDATE	ACCESS UPDATE

¹May be implied

Because the initial accessor can specify multiple SHARING values, a subsequent accessor whose ACCESS values match one, some, or all of the initial accessor's SHARING values is allowed access; however, when the subsequent accessor specifies an ACCESS value that the initial accessor did not specify as a SHARING value (an exception is SHARING GET, which is implied), access will be denied to the subsequent accessor.

In addition to the comparison of the file access values that subsequent accessors specify with the file-sharing values specified by the initial accessor, the values specified for file sharing by subsequent accessors must be compatible with the values specified for file access by the initial accessor. Table 7-4 shows the file-sharing values that subsequent accessors must specify to access the file.

Table 7-4 Initial File Access and Subsequent File Sharing

Initial Accessor Access	Subsequent Accessor Sharing
ACCESS GET ¹	SHARING GET ¹
ACCESS DELETE	SHARING DELETE
ACCESS PUT	SHARING PUT
ACCESS UPDATE	SHARING UPDATE
ACCESS TRUNCATE	No access allowed

¹May be implied

Because the initial accessor can specify multiple ACCESS values, a subsequent accessor whose SHARING values match all of the initial accessor's ACCESS values is allowed access; however, when the subsequent accessor specifies a SHARING value that the initial accessor did not specify as an ACCESS value (an exception is ACCESS GET which is implied), access will be denied.

Note that specifying ACCESS TRUNCATE disables file sharing.

7.1.3 User-Interlocked Interprocess File Sharing

The user-interlocked interprocess form of file sharing allows one or more users to write records to a sequential file residing on a disk device or to a file on a disk device that is open for block I/O processing. It cannot be used with relative and indexed files currently opened for record access. (For record access to relative and indexed files, VAX RMS transparently controls the reading and writing of buffers to the file, and always maintains current end-of-file information.)

All sequential files that reside on disk devices may be write shared with user-provided interlocks. To use this feature, you must specify SHARING USER_INTERLOCK (set the FAB\$B_SHR field FAB\$V_UPI bit). Note that when this option is specified, VAX RMS does not attempt to control the reading and writing of I/O buffers across processes, nor does it maintain end-of-file information. Thus, you must use the Flush service (or VAX language equivalent, if any) to force the writing of modified I/O buffers and to rewrite the record attributes (including end-of-file information) in the file header. Processes that open the file after that point will obtain the new end-of-file information. Note also that record attributes are rewritten whenever a file is closed. The last write accessor to close the file must also be the last accessor to have extended the file. If not, end-of-file information will be written by another write accessor. Read accessors of a shared sequential file can update their internal end-of-file context by closing and reopening the file.

No form of record locking is supported for this type of file sharing. Although record locking is not checked using user-interlocked interprocess file sharing, file locking is checked. For instance, if you or another user specify SHARING NONE, it is likely that either you or the other user will be denied access to the file.

If a process tries to implement the truncate service when closing a *sequential* file, it must have sole *write access* to the file. If other processes have *write access* to the file, VAX RMS will not close it and it remains accessible to other processes. If other processes have the file open for *read access*, VAX RMS defers the truncation until the final process having *read access* closes the file.

Similarly, if a process tries to implement the truncate-on-put option when inserting a record into a *sequential* file, it must have sole access to the file. If other processes have access to the file, VAX RMS will not insert the record.

7.2 Record Locking

VAX RMS provides a record-locking capability for all file organizations. This capability affords control over operations when two or more streams or processes are accessing the file simultaneously. Record locking ensures that when a program is adding, deleting, or modifying a record on a given stream, another stream or process cannot access the same record.

To prevent simultaneous updates of the same records, VAX RMS uses the VAX/VMS lock manager to lock a record that has been read and will be modified later by the same program.

VAX RMS can lock records in shared files to synchronize access to individual data items by different processes or streams, thus ensuring consistency of the data. This record-locking facility ensures that a program can add, delete, or modify a record on one stream without needing to check that the same record is not simultaneously accessed by another stream or process. For example, when your program opens a file to write or update records on multiple streams, VAX RMS locks each record as it is accessed by the program. This locking prevents another stream or process from accessing the same record until the first stream or process releases it.

VAX RMS also handles the automatic locking of the entire bucket that contains the record for the short period required to access the record initially. This automatic lock also occurs later, when the contents of the bucket are modified. In the interim, the record remains locked, but other records in the bucket can be accessed and modified as required.

You can request that VAX RMS automatically handle record locking and unlocking or you can request explicit control over the release of record locks.

Whether or not record locking can occur on a file depends on the file access and file sharing specified by the initial accessor and whether another user has successfully opened the file for shared access. Two combinations of file access and file sharing will allow record locking to occur when a file is accessed by multiple users for interlocked interprocess file sharing:

- The initial accessor specifies an ACCESS attribute that indicates record modification, such as ACCESS DELETE, ACCESS PUT, or ACCESS UPDATE in combination with any SHARING attribute that specifies interlocked interprocess file sharing, such as SHARING GET, SHARING DELETE, SHARING PUT, SHARING UPDATE, and/or SHARING MULTISTREAMING (but not SHARING PROHIBIT or SHARING USER_INTERLOCK).

- The initial accessor specifies ACCESS GET and a SHARING attribute that indicates that record modifications can occur, such as SHARING DELETE, SHARING PUT, and/or SHARING UPDATE. Even if all other processes have the file opened only for read access (ACCESS GET), record locking will be active.

Record locking can be switched off for a record stream on a per-operation basis (see Section 7.2.3 for information on the do-not-lock record option). Record locking can also occur in files opened only for read access when the lock-record-for-read option is set.

Records can be locked automatically or manually. VAX RMS handles automatic record locking transparently. You use it when you are dealing with a lock on a single record at a time. Manual record locking requires additional effort on your part. You use it when dealing with locks on multiple records at one time.

A record can be in one of three states: unlocked, automatically locked, or manually locked. When a record is locked initially, it is in either the manually or automatically locked state. It will remain in that state until the lock is released; the record cannot move directly from the automatically to manually locked state, or vice versa. Therefore, you must make an initial decision to use automatic or manual locking for a given record on the basis of your needs. You continue to use the same type of locking with that record until the lock is released.

7.2.1 Automatic Record Locking

When automatic record locking is used, each record returned or accessed by a Get or Find operation is locked (unless the do-not-lock record option is set). The lock is held until the record is completely processed by ensuing operations on the stream. That is, the lock is released when one of the following takes place:

- The next record is accessed.
- The current record is updated or deleted.
- The record stream is disconnected.
- The file is closed.
- An error occurs during a related operation.

File Sharing and Buffering

Thus, a record is automatically unlocked when you (or the language statement specified) invoke any of the following services:

Find	Locates a new record
Get	Locates and reads a new record
Put	Adds a new record
Update	Modifies the current record
Delete	Deletes the current record
Rewind	Positions the record stream to beginning of the file
Disconnect	Disconnects the record stream
Close	Closes the file and disconnects all record streams to it
Free	Releases locks on all records in the record stream
Release	Releases the lock on the current record

Note that the Free service and the Release service permit you to explicitly unlock the record.

If you use the Put service to write a record into an empty cell of a relative file, the cell is effectively locked during the Put service and is automatically unlocked when the service completes. There is one exception to the automatic unlocking: a record remains automatically locked on a sequential Get service that follows a Find service that locked the record.

The automatic record-locking scheme normally does exactly what is required without any interruptions. For example, you could use the following logical sequence to update an existing record:

- 1 Get the record (Get service)
- 2 Modify the record buffer (using language statements)
- 3 Update the record (Update service)

The Get service establishes the current record and locks it in preparation for the Update service. The program then operates on the record as required. When the record is finally updated, the record lock is released. In the interval between the Get and Update operations, other streams attempting to access the record receive a record-locked error (RMS\$_RLK). This prevents the original record from being accessed (and potentially modified) before the stream has finished operating on it. When the Update service releases the record, it becomes accessible to other streams.

7.2.2 Manual Record Unlocking

When unlocking records manually, you have explicit control over the unlocking of records. Thus, manual record unlocking lets you control operations that must be performed together.

To specify manual record unlocking, you must specify the CONNECT MANUAL_UNLOCKING attribute (RAB\$L_ROP field RAB\$V_ULK bit) when issuing a Get, Find, or Put service. (These three services will unlock any record that was previously locked with automatic record locking.) Once locked, the record will remain locked until unlocked by either the Free or Release service, or until the stream terminates (by a Disconnect or Close service). Other operations on the record or stream, including operations that result in errors, do not cause the record to be unlocked.

Manual control over the unlocking of records is useful when multiple records must be modified as a single transaction. An example of this would be when two separate records are randomly accessed and updated. The first record must not be accessed by another stream until modifications to the second record are complete. The program attempts to update the first record, but at the same time retains the lock on it. Thus, in the event of a failure to update the second record, the original contents of the first record could be restored. Manual unlocking is specified when accessing the first record. This will prevent the record from being unlocked automatically after the Update operation. The lock is released by using the Free service after successfully updating the second record (the normal case), or after restoring the original contents of the first record (the error condition). The Free service releases all locks for that stream simultaneously. At this time, both updated records will become accessible to other streams.

You can use the Release service selectively to release locked records, using the RFA value of the record.

7.2.3 Record Locking and Unlocking

VAX RMS provides several options for controlling record locking and unlocking. The following options can be specified as input to each Find, Get, or Put service.

Option	Description
Do not lock	Does not lock the record. FDL: CONNECT NOLOCK VAX RMS: RAB\$L_ROP RAB\$V_NLK
Lock nonexistent	Locks a nonexistent record (applicable to relative files, only). FDL: CONNECT NONEXISTENT_RECORD VAX RMS: RAB\$L_ROP RAB\$V_NXR
Lock for read	Locks the record for reading and allows other readers (but no writers). FDL: CONNECT LOCK_ON_READ VAX RMS: RAB\$L_ROP RAB\$V_REA
Lock for write	Locks the record for writing and allows other readers (but no writers). FDL: CONNECT LOCK_ON_WRITE VAX RMS: RAB\$L_ROP RAB\$V_RLK
Manual unlocking	Indicates that the user, not VAX RMS, will control record unlocking. Otherwise, VAX RMS automatically controls record unlocking for this record stream (unless the do-not-lock record option is set). FDL: CONNECT MANUAL_UNLOCKING VAX RMS: RAB\$L_ROP RAB\$V_ULK
Read regardless	Reads the record regardless of any lock. FDL: CONNECT READ_REGARDLESS VAX RMS: RAB\$L_ROP RAB\$V_RRL
Wait if locked	If the record is locked, wait until it is available. FDL: CONNECT WAIT_FOR_RECORD VAX RMS: RAB\$L_ROP RAB\$V_WAT
Wait timeout period	This option may be specified in conjunction with the wait-if-locked option to specify a timeout period after which an error is returned. The specified value indicates the specified number of seconds to wait before returning an error to avoid a potential deadlock. FDL: CONNECT TIMEOUT_PERIOD VAX RMS: RAB\$L_ROP RAB\$V_TMO and RAB\$B_TMO

See Section 7.2.2 for more information on manual record unlocking; the other options are discussed in detail below.

Do Not Lock Record

The do-not-lock record option specifies that the record accessed with either a Get or Find service is not to be locked. Specifying only Get file access for the file also implies that records are not to be locked. In either case, if the target record is locked by another stream, a record-locked error (RMS\$_RLK) will be returned from a direct VAX RMS service call. The only exception to this will be if the stream locking the record has allowed readers (see the lock-record-for-write or lock-record-for-read options below). Records accessed for purposes other than modifying (that is, deleting or updating) should not be locked. This will reduce the probability that other streams will receive record-lock errors. Attempts to delete or update a record that was not locked will fail. The do-not-lock record option takes precedence over the manual-record-unlocking option.

Lock Nonexistent Record

The lock-nonexistent-record option applies only to relative files. It is used to lock randomly accessed records that do not exist in the file at the time of access. This prevents other streams from putting a new record into that cell until the stream that locked it either puts a record there itself or releases the record lock. For example, suppose that a file contains records 1 through 10. A program attempting to randomly access record 15 in a relative file would normally receive a record-not-found (RMS\$_RNF) error. If the lock-nonexistent-record option is specified, VAX RMS returns an alternative success code indicating successful access of a nonexistent record (RMS\$_OK_RNF). Any other stream attempting to access or put data into record 15 receives a record-locked error (RMS\$_RLK). If the stream that locked the nonexistent record attempts to put a new record there, VAX RMS returns an alternate success code indicating that the record was already locked (RMS\$_OK_ALK).

This option may also be used to randomly access (by relative record number) the contents of a deleted record from a relative file. For example, if you requested to read record 10 in a relative file and that record had been deleted, VAX RMS would return a record-not-found message (RMS\$_RNF). However, if you had specified the lock-nonexistent-record option when requesting that record, the contents of that deleted record would be returned along with the alternate success status (RMS\$_OK_DEL) to indicate that a deleted record had been accessed.

Lock Record for Read

The lock-record-for-read option specifies a lock for reading that allows other readers but no writers. If this option is specified, then a record stream with a shared file that is open for read only, is permitted to lock records from modification by other programs or streams. When a nonlocking stream reads a record locked in this manner, an alternate success code (RMS\$_OK_RLK) indicates that the target record was indeed locked, but that readers are allowed. Another stream attempting to lock the record, however, will still receive a record-locked error (RMS\$_RLK).

Lock Record for Write

The lock-record-for-write option specifies that the record will be locked for possible modifications. However, readers will be able to access the record. Streams that are locking records for modification may allow nonlocking streams to read locked records. When a nonlocking stream reads a record locked in this manner, an alternate success code (RMS\$_OK_RLK) indicates that the target record was indeed locked, but that readers are allowed. Another stream attempting to lock the record, however, will still receive a record-locked error (RMS\$_RLK).

Read Regardless of Lock

The read-regardless-of-lock option gives a reading process some control over record access by permitting it to read the record regardless of whether or not it is locked. If a Put or Get operation tries to process a record that is locked against all access, this option will return the record with the alternate status RMS\$_OK_RRL.

Wait for Record

The wait-for-record option specifies that if the record is already locked, your process will wait until the record is available. When the requested record is available, it will be returned with the status RMS\$_OK_WAT.

When two separate processes have each specified the wait-for-record option and the manual-unlocking option, and both processes attempt to access the same records in a file, the possibility of a deadlock arises. For example, if process A and process B wish to access records 1 and 2, at the same time, the following situation could occur. Process A could have locked record 1 and specified that it would wait for record 2. At the same time, process B could have locked record 2 and specified that it would wait for record 1. In this case, VAX RMS would return an RMS\$_DEADLOCK error (signaling the deadlock) and ignore one of the wait requests.

The amount of time that will elapse before you are notified of the deadlock will depend on the value specified in the DEADLOCK_WAIT system parameter. The default value for this system parameter is 10 seconds. Thus, 10 seconds will elapse between the occurrence of the deadlock and the initiation of the search in behalf of the deadlock. For further details on how this parameter is set, see the *VAX/VMS System Generation Utility Reference Manual*.

Wait Timeout Period

This option specifies that when the wait for record option is set, a record-wait condition exists only for the specified timeout period, in seconds. If the timeout period expires before the lock is granted, VAX RMS ignores the request and returns an RMS\$_TMO completion status. You set this timeout period by specifying the number of seconds for the CONNECT TIMEOUT_PERIOD attribute, or by directly setting the RAB\$_TMO bit (in the RAB\$_ROP field) and storing the number of seconds to wait in the RAB\$_TMO field.

7.2.4 Programming Techniques for Shared Files

Applications that are designed to process shared files must be ready to handle record-locking conflicts that may occur when two or more record streams try to access the same record. When a conflict occurs, an RMS\$_RLK error is returned to the record stream.

There are two different techniques you can use to deal with record-locking conflicts. You can design your program to wait for the record to become available or have the program try again to access the record.

This section briefly describes these techniques.

7.2.4.1 Waiting for Records

To design your program to wait for records, you should specify the wait-for-record option. The program will not proceed until it has gained access to the record and you will not receive a record-locking error. The programming task is often simplified by using the wait-timeout-period option in combination with the wait-for-record option.

In some cases, you may have to be concerned with deadlock errors that can occur. If you have multiple record streams or multiple records locked by a single stream, then you cannot simply try to access a record again after a deadlock error is returned. If you do, the deadlock will continue. Your program must use either the Free or the Release service to break the deadlock by releasing all locked records, and then go back and reaccess each record it needs.

The outline of program logic below shows this technique for updating two records in a shared file. The wait-for-record option is specified.

- 1 Specify the wait-for-record, wait-timeout-period, and manual-record-unlocking options (RAB\$L_ROP field, RAB\$V_TMO, RAB\$V_WAT, and RAB\$V_ULK bits; and the RAB\$B_TMO field)
- 2 Find first record; branch on error (end program)
- 3 Find second record; branch on success (process records)
- 4 Check completion status for RMS\$_DEADLOCK
- 5 If error is not RMS\$_DEADLOCK, branch unconditionally (end program)
- 6 If RMS\$_DEADLOCK, use the Free service to free all record locks for this record stream
- 7 Repeat the procedure until the operation completes without a deadlock.

7.2.4.2 Try Again on Record Lock Errors

Another technique that you can use to deal with record-locking conflicts when processing shared files is to design your application program to try to access the record again.

If the process that owns the record lock finishes and releases the record before your program tries again to access the record, your program will be able to access the record and proceed normally. However, your process must avoid using system resources that the owner process needs to complete its operation. Typically this happens if your process continuously executes Get or Find services in attempting to access the locked record and results in either a processing deadlock or very slow progress by the owner process.

Therefore, applications that are designed to retry for record access on a record-lock error should be designed to wait (or delay) for a short time before attempting to reaccess the record through a Get or Find operation. The wait or delay will enable the owner process to complete the processing of the record. This processing deadlock problem can be particularly serious if the shared file being processed is using global buffers.

Example 7-1 contains a program fragment written in VAX MACRO that demonstrates one method of implementing a short wait before attempting to access the record again.

This example uses an event flag for synchronizing the delay. Other techniques, such as hibernation or ASTs, may also be used. See the *VAX/VMS System Services Reference Manual* for more information on these process control techniques.

Example 7-1 Example of Waiting to Access a Locked Record

```

ONE_SECOND:
    .LONG    -10000000,-1        ; 1 second delta time
    .
    .
10$:    $GET     RAB=INRAB          ; Get the record
        BLBS    RO,GOT_RECORD     ; Branch on success
        CMPL    RO,$RMS$_RLK      ; Record-locked error?
        BNEQ    ERROR             ; Quit on other errors
        $SETIME_S      EFN=#1,-    ; Set event flag
        DAYTIM=ONE_SECOND         ; In one second
        BLBC    RO,ERROR          ; Quit on error
        $WAITFR_S      EFN=#1     ; Wait for event flag
        BRB     10$              ; Try again for record
    .
    .

```

7.3 Local and Shared Buffering Techniques

One of the key performance factors is record buffering, that is, the transfer of records between a storage device and an area of memory accessible to the user's program. Between the storage device and the record buffer in the user's program, however, is an intermediate buffer area that VAX RMS maintains. An intermediate buffer area is usually associated with each process; you can also specify a shared buffer area if the file will be shared.

7.3.1 Record Transfer Modes

For synchronous and asynchronous record operations, VAX RMS provides two record transfer modes: move mode and locate mode.

In move mode, VAX RMS copies a record from an I/O buffer into a buffer that you have specified. For input operations, data is first read into the I/O buffer from a peripheral device (such as a disk), and then moved to your user buffer for processing. For output operations, you first build the record in your buffer; VAX RMS then moves the record to an I/O buffer. When an entire block is filled in the I/O buffer, the block is written to the peripheral device.

In locate mode, VAX RMS allows the user program to access records directly in a VAX RMS I/O buffer by providing the address of the returned record as the internal VAX RMS buffer location instead of a user buffer location (field `RAB$L_RBF`). Usually, this reduces program overhead because records can be processed directly within the I/O buffer. Locate mode is only available for input operations. Because it may not always be possible to use locate mode, you must supply a user buffer for cases in which move mode must be used, even though you may have specified locate mode (see the *VAX Record Management Services Reference Manual*).

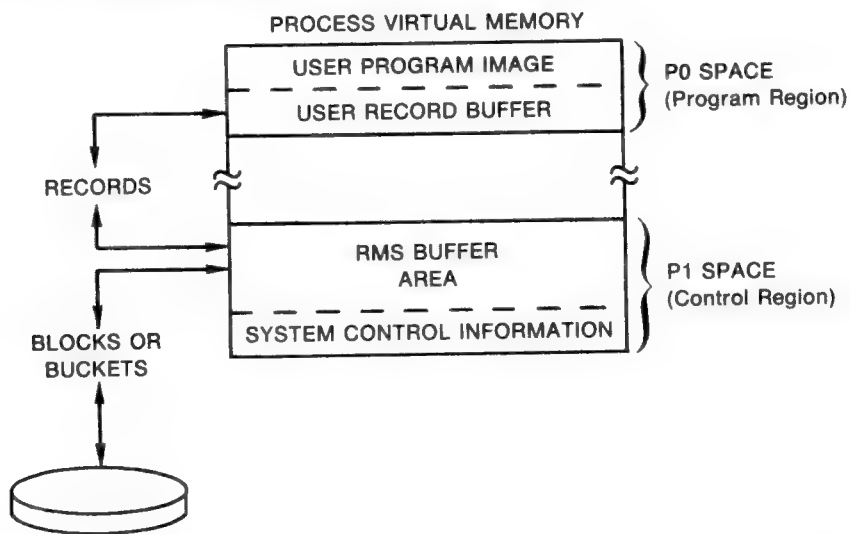
Other VAX RMS facilities allow programs to control I/O buffer space allocation or to simply leave all space management to VAX RMS. The following sections briefly describe the file-level and record-level processing environments, and the different modes of file access that a program can use with VAX RMS.

7.3.2 Understanding Buffering

Record processing under VAX RMS appears to your program as the movement of records directly between a file and the program itself. This is, in fact, not the case. VAX RMS uses internal memory areas called I/O buffers to read or write blocks or buckets of data. Transparently to your program, VAX RMS transfers blocks or buckets of a file into or from an I/O buffer. Records within the I/O buffer are then made available to the program when VAX RMS transfers the records between the I/O buffer and the user's record buffer.

The unit of data transfer between a file and the I/O buffers depends on the file organization. For the sequential organization, VAX RMS reads and writes a block or series of blocks. For relative and indexed organizations, VAX RMS reads and writes buckets.

The relationship between the user program and the I/O buffers that VAX RMS maintains is shown in Figure 7-2. As illustrated, the user program resides in the P0 region of process address space and the VAX RMS-maintained buffer area, together with VAX RMS-maintained control information, is located in the P1 region. Note that linker options are available for allocating additional buffer space in the P0 region, if needed. See the *VAX/VMS Linker Reference Manual* for details.

Figure 7-2 VAX RMS Buffers and the User Program

ZK-1993-84

The specified record buffer contains the record to be read or written, and VAX RMS maintains the rest of the block in user process space in a VAX RMS-controlled area of the program.

When performance is important, it is recommended that the number of buffers be considered carefully. The defaults calculated by VAX RMS are small in number and this is often adequate for certain types of access to small files. For example, it is not unusual to specify many buffers when processing a large indexed file, yet VAX RMS provides a default of only two buffers for indexed files.

The CONNECT secondary attribute MULTIBUFFER_COUNT (VAX RMS field RAB\$B_MBF) establishes the number of buffers, but the FILE secondary attribute GLOBAL_BUFFER_COUNT (VAX RMS field FAB\$W_GBC) specifies the number of *global buffers* as described below.

Often the best ways to achieve optimum buffering for a particular application is to test various combinations of buffer sizes with numbers of buffers. One approach is to time each combination and measure the number of I/O operations that take place, and then choose the one that improved application performance the most considering the amount of memory used.

The general idea of using buffering to improve performance is to use a buffer size and number of buffers that improves application performance without exhausting the virtual memory resources of your process or system. Keep in mind the trade-offs between file I/O performance and exhausting memory resources. The buffers used by a process are charged against the process's working set. You should avoid allocating so many buffers that the CPU spends excessive processing time paging and swapping. For performance-critical applications, consider increasing the size of the process working set and adding additional memory.

The system manager should monitor the paging and swapping activity of the application's process and selected other processes to avoid improving the performance of the target application at the expense of other applications. Have your system manager consult the *Guide to VAX/VMS Performance Management* for system tuning information. Refer to Section 1.6 for information on the resources needed for file applications.

When records are likely to be accessed sequentially, a large buffer (or buffers) should be used. The reasoning behind this is that the physically contiguous records in a file are read into memory in one or more blocks for sequential files or in buckets (multiblock units) for relative and indexed files. Once the blocks or buckets are read into the buffer area provided by VAX RMS, subsequent access to adjacent records would access records in the same block or bucket that are already in the buffer, eliminating the need for an additional I/O, thus improving performance. When a record is needed that is not in the current buffer cache, one of the buffers is replaced by the block(s) or bucket that contains the new record.

When records in the file will be reused (such as the index of an indexed file), using more than one buffer can hold the previously accessed records in memory for a longer period of time to eliminate another I/O operation when those records are accessed again later. There is also a special buffering option for sequential files where two buffers are used alternately, allowing one buffer to be used while the other awaits I/O completion.

The buffers that the application requests VAX RMS to allocate for its use are referred to as a *buffer cache* and can be thought of as a buffer pool for your process that VAX RMS uses to locate records first before attempting I/O to the target device. When a file is shared among many processes, a different type of buffer cache is available that can be shared among multiple processes.

7.3.3 Buffering for Sequential Files

With sequential files, the number of buffers and the size of the buffers can be specified at run time. The number of buffers is specified by the FDL attribute `CONNECT MULTIBUFFER_COUNT` (VAX RMS control block field `RAB$B_MBF`) and the buffer size is specified by the FDL attribute `CONNECT MULTIBLOCK_COUNT` (VAX RMS control block field `RAB$B_MBC`).

The use of sequential files provides an option that allows the use of two buffers that are used alternately to contain the next records to be read or written to the disk while the other buffer awaits I/O completion. This is called read-ahead and write-behind processing and should be considered for sequential access to sequential files. The number of buffers (`CONNECT MULTIBUFFER_COUNT`) should be specified as 2. The length of the buffers used for sequential files is determined by the specified multiblock count (`CONNECT MULTIBLOCK_COUNT`). For sequential access to a sequential file, the optimum number of blocks per buffer depends on the record size, but a value such as 16 is usually appropriate.

When accessing a sequential file randomly (using RFA access), refer to the following discussion for relative files.

To see what the current process-default buffer count is, use the DCL command `SHOW RMS_DEFAULT`. To set the process-default buffer count, use the DCL command `SET RMS_DEFAULT/SEQUENTIAL/BUFFER_COUNT=n`, where `n` is the number of buffers.

7.3.4 Buffering for Relative Files

With relative files, buckets, not blocks, are the unit of transfer between the disk and memory. The bucket size is specified when the file is created, although the bucket size of an existing file can be changed by converting the file (see Chapter 10).

The bucket size is specified by the FDL attribute `FILE BUCKET_SIZE` (VAX RMS control block field `FAB$B_BKS` or `XAB$B_BKZ`). When choosing this value, you should consider whether or not the file will usually be accessed randomly (small bucket size), sequentially (large bucket size), or both (medium bucket size), as described in Chapter 2.

The number of buffers (`CONNECT MULTIBUFFER_COUNT`, `RAB$B_MBF`) is specified at run time. The type of record access to be performed determines the best use of buffers. The two extremes of record access are that records will be processed completely randomly or completely sequentially. Also, there are cases where records are accessed randomly but may be reaccessed (random with temporal locality) and cases where records are accessed

randomly but adjacent records are likely to be accessed (random with spatial locality).

For completely random or for sequential access, a single buffer should be specified. In a processing environment where the program is processing records randomly and sometime reaccessing records, it is recommended that you use multiple buffers to keep the records that will soon be reaccessed in the buffer cache.

When records are accessed randomly and adjacent records are apt to be accessed, you should specify a single buffer. However, if your program is processing a file with small bucket sizes, you should consider specifying more buffers. When the file is likely to be accessed by different combinations of the above, a compromise of the number of buffers (and bucket sizes) is recommended.

When adding records to a relative file, consider choosing the deferred-write option (FDL attribute `FILE DEFERRED_WRITE`; `FAB$L_FOP` field `FAB$V_DFW`). With this option, the buffer (memory-resident bucket) into which the records have been moved is not written to disk until the buffer is needed for other purposes or the file is closed. This option, however, may cause records to be lost if a system crash should occur before the records are written to disk.

To see what the current process-default buffer count is, use the DCL command `SHOW RMS_DEFAULT`. To set the process-default buffer count, use the DCL command `SET RMS_DEFAULT/RELATIVE/BUFFER_COUNT=n`, where `n` is the number of buffers.

7.3.5 Buffering for Indexed Files

With indexed files, buckets (not blocks) are the unit of transfer between the disk and memory. The bucket size is specified when the file is created, although the bucket size of an existing file can be changed by converting the file (see Chapter 10).

The bucket size is specified by the FDL attribute `FILE BUCKET_SIZE` (VAX RMS control block field `FAB$B_BKS` or `XAB$B_BKZ`), as described in Chapter 2.

When accessing indexed files, it is important to remember that the index portion of the file must be read by VAX RMS to locate the desired record. The algorithm used by VAX RMS places a higher priority for the higher-level buckets of the index in the buffer cache. Thus, the highest levels of the index remain in the buffer cache, while the buffers that may have contained the actual data buckets and the lower-level index buckets are reused to contain other buckets. That is, the buffers that are reused first contain either data or

lower-level index buckets, which are the first to be discarded from the buffer cache.

The number of buffers (CONNECT MULTIBUFFER_COUNT, RAB\$B_MBF) is specified at run time and recommended values can vary greatly for different applications when accessing indexed files. The following suggestions on use of buffers apply to the type of record access to be performed:

- Completely random processing—When records are processed randomly, the use of as many buffers as your process working set can support is recommended to cache as many index buckets as possible.
- Sequential processing—When records will be accessed sequentially, even after locating the first record randomly, the use of a small multibuffer count, such as the default of 2 buffers, is sufficient.

Many applications will access files using a mixture of completely random and completely sequential processing. For such applications, a compromise of the above number of buffers is recommended.

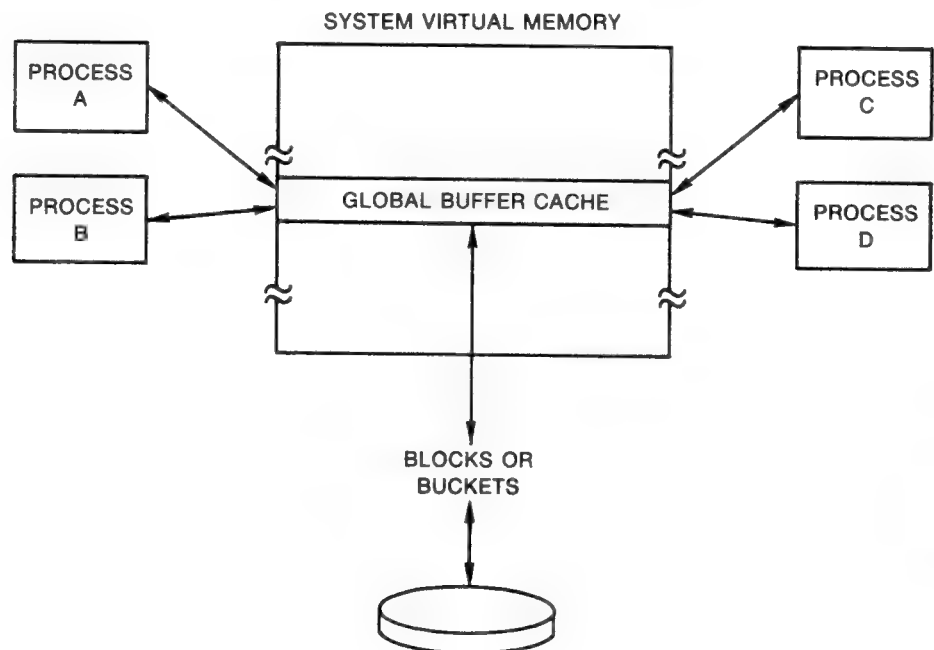
When adding records to an indexed file, consider choosing the deferred-write option (FDL attribute FILE DEFERRED_WRITE; FAB\$L_FOP field FAB\$V_DFW). With this option, the buffer into which the records have been moved is not written to disk until the buffer is needed for other purposes or the file is closed. This option, however, may cause records to be lost if a system crash should occur before the records are written to disk.

To see what the current process-default buffer count is, use the DCL command SHOW RMS_DEFAULT. To set the process-default buffer count, use the DCL command SET RMS_DEFAULT/INDEXED/BUFFER_COUNT=n, where n is the number of buffers.

7.3.6 Using Global Buffers for Shared Files

Two types of buffer caches are available using VAX RMS: local and global. Local buffers reside within process (program) memory space and are not shared among processes, even if multiple processes are accessing the same file and reading the same records. Global buffers, which are designed for applications that access the same files and perhaps may even access the same records, do not reside in process memory space.

If several processes will share a file, you should specify that the file will use global buffers. A global buffer is an I/O buffer that two or more processes can access in conjunction with file sharing. If two or more processes are requesting the same information from a file, each process can use the global buffers instead of allocating its own process-local buffers. Figure 7-3 illustrates the use of global buffers.

Figure 7-3 Using Global Buffers for a Shared File

ZK-1994-84

Unlike local buffers, global buffers can be accessed by multiple processes accessing the same file. When a record requested by one process is located in one of the global buffers, the record can be transferred directly from the global buffer to the program, eliminating a disk I/O operation. If the record contained in the global buffer was written by a process other than the process requesting the record, VAX RMS writes the contents of the buffer to disk before returning the record to ensure that the modified bucket matches its counterpart on the disk.

There are two situations in which global buffers cannot be used for shared files. When a process permanent file is being accessed, VAX RMS will not use global buffers (no error is returned). When an image is linked using the LINK option keyword IOSEGMENT-NOP0BUFS (rarely used), VAX RMS will not use global buffers.

Even if global buffers are used, a minimal number of local buffers should be requested, because, under certain circumstances, VAX RMS may need to use local buffers. When attempting to access a record, VAX RMS looks first in the global buffer cache for the record before looking in the local buffers; if the record is still not found, an I/O occurs. When using deferred write with global buffering enabled, the number of buckets that can be buffered without I/O will be equal to the number of local buffers; thus, the use of more than the minimum number of local buffers should be considered.

The number of global buffers used is specified in one of two ways: by using a preset file default or by having the first process that accesses the file specify the value at run time. To set the file default (maintained in the file header), use the DCL command `SET FILE/GLOBAL_BUFFER=n`.

To set the value at run time, the first process to connect to the file with the `FILE GLOBAL_BUFFER_COUNT` attribute (VAX RMS control block field `FAB$W_GBC`) greater than 0 can set this value. The default value returned in the `FAB$W_GBC` field following an Open (or Create) service may be altered if unacceptable before invoking the Connect service. When a previous or subsequent application attempts to open and connect to the file, the global buffer count determines whether or not that process uses global buffers. If the value is 0, that process uses only local buffers; if the value is greater than 0, that process uses global buffers along with other processes. Refer to the *VAX Record Management Services Reference Manual* for additional information on the use of the `FAB$W_GBC` field and Connect service. An example of a routine that sets the global buffer count after opening a file is provided in Example 5-2.

To request that the global buffer cache be a read-only global buffer cache, specify `SHARING GET` and `SHARING MULTISTREAMING` attributes (`FAB$B_SHR` field `FAB$V_SHRGET` and `FAB$V_MSE`). This will improve performance by eliminating certain internal operations, such as the maintenance of bucket locks in the global buffer cache.

Note that when using global buffers for an indexed file, using more than the suggested number of buffers stated previously for the number of local buffers is recommended (the number depends on the file organization and the type of record access). When modifying an application to use global buffers, you might consider experimenting with more global buffers (and/or slightly larger bucket sizes) when records will be processed randomly. For applications with many users, consider allocating as many global buffers as there were local buffers previously used, multiplied by the approximate average number of users (if resources permit). When using an indexed file, if the index structure is small and the number of users is many, consider allocating enough global buffers to keep the entire index structure in memory.



8

Record Processing

This chapter introduces you to record processing to help you in using the various run-time record operations described in Chapter 9. Here you will find information related to the following subjects:

- Record operations appropriate to various high-level languages
- Accessing records for the various file organizations
- Record environment in terms of record positioning
- Synchronous versus asynchronous record operations

8.1 Record Operations

Record operations are performed by VAX RMS services which are classified further as being either primary or secondary services. The distinction is that primary services are functionally similar to high-level language record operations whereas secondary services are functionally peculiar to VAX RMS.

Section 8.2 describes the five primary services; refer to Section 8.3 for a brief description of the secondary services and the *VAX Record Management Services Reference Manual* for more detailed descriptions of the secondary services.

8.2 Primary VAX RMS Services

This section describes the five VAX RMS services that are functionally similar to related high-level language operations. The following list provides a brief description of each of these services together with references to the functional similarities of high-level languages.

- Find** This service locates an existing record in the file. It does not return the record to your program; instead it establishes the record's location as the current-record position in the record stream. The Find service, when applied to a disk or magnetic tape file, corresponds to the FIND statement in BASIC and FORTRAN, the START statement in COBOL, the FIND or LOCATE statements in PASCAL, and the READ statement with the SET keyword for PL/I.

Record Processing

- Get This service returns the selected record to your program. The Get service, when applied to a disk or magnetic tape file, corresponds to (is used by) the GET statement in BASIC; the READ statement in COBOL, FORTRAN, and PL/I; and the GET statement (and others) in PASCAL.
- Put This service inserts a new record in the file. The Put service, when applied to a disk or magnetic tape file, corresponds to the PUT and PRINT statements in BASIC; the WRITE statement (and others) in COBOL; the WRITE statement in FORTRAN and PL/I; and the PUT and WRITELN statements in PASCAL.
- Update This service modifies an existing record. The Update service, when applied to a disk file (it can only be used for disk files), corresponds to the UPDATE statement in BASIC and PASCAL and to the REWRITE statement in COBOL, FORTRAN, and PL/I.
- Delete This service deletes an existing record from a file. The Delete service, when used for disk files (it can only be used for relative and indexed disk files), corresponds to the DELETE statement in BASIC, COBOL, FORTRAN, PASCAL, and PL/I.

A single statement in a VAX language may correspond to one or several VAX RMS record-processing service calls. For example, the COBOL statement DELETE uses the VAX RMS Delete service during sequential record access, but uses the Find and Delete services during random record access.

File organization in part determines the types of record operations that a program can perform. Table 8-1 shows the major record operations that VAX RMS permits for each file organization.

Table 8-1 Record Operations and File Organizations

Record Operation Permitted	Sequential	File Organization	
		Relative	Indexed
Get	Yes	Yes	Yes
Put	Yes ¹	Yes	Yes
Find	Yes	Yes	Yes
Delete	No	Yes	Yes
Update ²	Yes ³	Yes	Yes

¹In a sequential file, VAX RMS allows records to be added at the end of the file only. (Records can be written to other points in the file using an Put service with the update-if option.)

²When performing an Update service to a sequential file, you cannot change the length of the record.

³VAX RMS allows Update services on disk devices only.

The remainder of this section briefly describes the record retrieval (Find and Get) services, the record insertion (Put) service, the record modification (Update) service, and the record deletion (Delete) service. Note that all references to VAX RMS services imply applicability to similar functional capabilities found in high-level languages.

8.2.1 Locating and Retrieving Records

You can use the Find and Get services to locate and retrieve a record. The Find service will locate a record and establish its location as the current-record position in a record stream but does not return the record to a buffer. The Get service locates the record, establishes its location as the current-record position in the record stream and returns it to the buffer area you specify.

If you use the Get service, you must preallocate a buffer area in the data portion of your program to store the retrieved record by defining an appropriate variable or multivariable record structure in the program. In addition to retrieving the record, VAX RMS returns to your program the length of the record (in control block field RAB\$W_RSZ, record size) and the file address of the record (in control block field RAB\$L_RBF, record buffer). If your program directs it to simply locate the record, VAX RMS does not write the record into your buffer. Instead, it sets the RAB\$W_RSZ and RAB\$L_RBF fields to point to an internal buffer where the record is located.

When using indexed files, you may need to allocate a buffer for the desired key and to specify its length. When using high-level VAX languages, the language's compiler may automatically handle the allocation and size specification of the record buffer and the key buffer.

There are situations where you can minimize record I/O and improve performance by using the Find service instead of the Get service. For example, a process need not retrieve a record when it is preparing to invoke the Update, Delete, Release, or Truncate service. If a process intends to update a record that is accessible to other processes, it should lock the record until it completes the update.

For interactive applications where the user verifies that the appropriate record is being accessed before deleting it or updating it, the program should use the Get service instead of the Find service.

In some situations, a process may use two services and two types of record access to retrieve a set of records. For example, the process might use the Find service and random access mode to locate the first record in the set, and then switch to the Get service for sequentially retrieving the records in the set.

An efficient use of the Find service is to create a table of RFAs (record file addresses) to be used for rapidly accessing the records in the same file.

Record retrieval operations are typically used to repetitively read and process a set of records. As part of this type of operation, your program should check for an end-of-file condition after each Find or Get service.

Refer to the *VAX Record Management Services Reference Manual* for more information on the Find and Get service.

8.2.2 Inserting Records

The Put service adds a record to the file. Within the data portion of your program, you must provide a buffer for the record that is to be added. The program must also supply the length of each record to be written when calling VAX RMS directly. This will be a constant value with fixed-length records but will vary from record to record when adding variable-length or VFC records. When using high-level VAX languages, however, the language's compiler may automatically handle record buffer size specification or supply a means to simplify its specification.

The current-record position is especially important when adding records to a sequential file. VAX RMS establishes the current-record position at the end of file for any record stream associated with a file opened for adding records. To add records to a relative file or to an indexed file, use random access (by key or record number), unless the program adds records sequentially by a specified ordering of primary keys or by relative record number.

The update-if option allows the record to replace an existing record, if one already exists, using the Put service when random access has been chosen. You might use this option when adding records to a relative or indexed file when a record-already-exists error is encountered and the new record supersedes the existing record. The update-if option also allows the program to update a record in a sequential file being accessed randomly by relative record number. When using this option for a shared file, use care with automatic record locking because the Put service, unlike the Update service, will briefly release record locks (that should have been applied using the Get or Find service before the Update operation) until it is converted into an Update service. This could allow another record stream to delete or update the record between the time that the Put service is invoked and the time when the Put service is converted to an Update service. Your program should use the Update service instead of the Put service with the update-if option to update an existing record in a shared file.

When a file contains alternate keys with characteristics that prohibit duplicate values, the application must be prepared to handle duplicate-alternate-key errors.

Refer to the *VAX Record Management Services Reference Manual* for more information on the Put service.

8.2.3 Updating Records

The Update service modifies an existing record in a file. Your program must first locate the appropriate record and optionally retrieve the record itself, by either calling the Find or the Get service. As with the Put service, your program must provide a buffer within the data portion of the program to hold the record that is to be updated.

The program must also supply the length of each record to be written when calling VAX RMS directly. This will be a constant value when updating fixed-length records but will vary from record to record when updating variable-length records or VFC records. Note that some high-level VAX language compilers may automatically handle record buffer allocation and size specification, or may supply a means to simplify its specification.

Your program must establish the current-record position before it updates a record. If the file is shared, the service that establishes the record position should also lock the record.

When you update indexed file records, take care not to alter the value of any key field that has been specified as unchangeable, for example, the primary key. To change the value of a record's primary key, you must replace the existing record with a new record having the desired primary key value.

You can do this using the Put and Delete services respectively, or, where applicable you may use the Put service with the update-if (RAB\$L_ROP RAB\$V_UIF) option.

When updating records in an indexed file, a key of reference does not need to be specified.

Refer to the *VAX Record Management Services Reference Manual* for more information on the Update service and record-processing options.

8.2.4 Deleting Records

This service deletes a record from the file. You cannot delete individual records from sequential files, but you can truncate them using the Truncate service. As with the Update service, the Delete service must be preceded by a Find or Get service to establish the current-record position.

When deleting records from an indexed file with alternate indexes, you can specify the fast-delete option to reduce the amount of time needed to delete a record. When you invoke the Delete service and specify fast delete, VAX RMS does not attempt to remove any of the pointers from alternative indexes to the deleted record.

You improve performance by postponing the processing needed to eliminate the pointers from alternative indexes to the record. However, there is a cost associated with this option. First, the unused pointers from the alternate indexes result in a corresponding waste of space. Second, if the program later tries to access the deleted record from an alternate index, VAX RMS must traverse the pointer linkage, find the record no longer exists and then perform the processing that was avoided originally with the Delete service.

You should consider this option only if the immediate improvement in performance is worth the added space and overhead. Typically, you would consider the fast-delete option for indexed files that implement alternate keys and require frequent maintenance.

Conversely, you should avoid this option for:

- most read-only indexed files.
- indexed files that are infrequently updated.
- applications that employ non-standard use of alternate keys. That is, applications where alternate keys are not used to access records, but instead act as a gauge for monitoring file content.

Refer to the *VAX Record Management Services Reference Manual* for more information on the Delete service.

8.3 Secondary Services

This section provides very brief descriptions of the VAX RMS secondary services. Note that each of the services performs a specialized function for which there are few options.

Connect	VAX RMS allows you to connect to a single record stream or to multiple record streams.
Disconnect	VAX RMS allows you to disconnect a record stream. This is done implicitly when a file is closed, but when using multiple record streams, you may want to disconnect one record stream but not others.
Flush	Writes modified I/O buffers and file attribute information maintained in memory to the file.
Free	Releases all record locks on records locked by this record stream.
Next Volume	Used for magnetic tape files to continue the next volume of a magnetic tape volume set. This service applies only to sequential tape files.
Release	Releases the record lock on the current record.
Rewind	Positions the record stream context to the first record of the file.
Truncate	This service truncates a file beginning with the current record, effectively deleting it and all remaining records. This service is applicable to sequential files only.
Wait	Awaits the completion of an asynchronous record operation (or Connect service).

In addition to the record processing services, a variety of file-processing services are also available. See the *VAX Record Management Services Reference Manual* for more information about both types of processing services and the options that apply to each.

8.4 Record Access for the Various File Organizations

To retrieve or insert a file record for a particular record stream, your program must specify either sequential or random access.

Sequential access can be used with all file organizations. For files organized sequentially, sequential access implies that records are accessed according to their physical position in the file. For relative files, sequential access implies that records are accessed according to the ascending order of relative record numbers. In indexed files, sequential access implies that records are accessed according to a specified ordering of values for a particular key or keys.

Random access is further defined as one of the following:

- Random-by-key access for indexed files implies that VAX RMS uses the specified key value (contained within the record itself) to locate the desired record.
- Random-by-relative-record-number access for relative files and for sequential files having fixed-length records, implies that the specified relative record number is used to locate the desired record. The relative record number does not necessarily reside in the record.
- Random-by-RFA access implies the specified record's file address (RFA) is used to locate the desired record. This access mode is supported for all three file organizations and is normally available only to programs written in VAX MACRO or similar low-level VAX languages.

Record access is specified using language statements or by establishing the appropriate control block field values (not offsets) in the RAB.

Note

There are no FDL attributes provided for specifying record access.

The appropriate RAB values in the access mode specification field, identified by the symbolic offset RAB\$B_RAC, are listed below.

- You specify sequential access by inserting the value RAB\$C_SEQ in the RAB\$B_RAC field.
- You specify either random-by-key access or random-by-relative-record-number access by inserting the value RAB\$C_KEY in the RAB\$B_RAC field. This access mode is used to randomly access records in an indexed files using a specified key value. It is also used to randomly access records by record number in relative files and in sequential files having fixed-length records.
- You specify random-by-RFA access for all file organizations by inserting the value RAB\$C_RFA in the RAB\$B_RAC field.

Your program may also need to specify the key or other record identifier needed to access the records. For indexed files, there are additional key-related options.

The record access mode can be changed without reopening the file or reconnecting the record stream. For example, you can use random-by-key access to establish the current-record position in an indexed file and then retrieve records sequentially by a specified sort order. Note however, that changing modes in this manner requires program access to the appropriate control block field at run time.

The record access mode, in conjunction with the file organization, is what determines the manner in which a record is selected. In the following sections, the sequential and random access modes are discussed in the context of the applicable file organizations. Random-by-RFA access is discussed separately because it applies to disk files, regardless of file organization.

The following discussion of the record-access modes is directed primarily toward the record-inserting/record-retrieving services described in greater detail in the *VAX Record Management Services Reference Manual*.

8.4.1 Processing Sequential Files

A program can read sequential files on both tape and disk devices using the sequential record-access mode. If the file resides on disk, the random-by-RFA record access mode can be used to read records; and if the file uses the fixed-length record format, the random-by-relative-record-number access mode is permitted.

You can add records only at the end of a sequential file.

All record access modes permit you to establish a new current-record position in a sequential file using the Find service. With sequential access, the Find service permits you to skip over records and with either random-by-relative-record-number or random-by-RFA access, the Find service establishes a starting point for sequential Get services.

You cannot randomly delete records from a sequential file. However, you can randomly update records in a sequential file if the file is on disk and if the update does not change the record size.

The following sections discuss the use of sequential and random access modes with sequential files.

8.4.1.1 Sequential Access

The sequential access mode is supported for sequential files on all devices. It is the only record access mode that is supported for nondisk devices, such as terminals, mailboxes, and magnetic tapes.

With sequential access, VAX RMS returns records from sequential files in the order in which they were stored. When a program has retrieved all of the records from a sequential file, any further attempt to sequentially access records in the file causes VAX RMS to return an end-of-file (no more data) condition code.

In sequential access mode, you can add records only to the end of a sequential file, that is the file location immediately following the current-record position.

8.4.1.2 Random Access

You can use the relative record number to randomly retrieve and insert records in sequential files having fixed-length records. Records are numbered in ascending order, starting with number 1.

In a sequential file, records are usually inserted at the end of the file. If you wish to randomly insert records within the current boundaries of the file (at a relative record number that is less than or equal to the highest record number in the file), you must set the update-if option (FDL attribute CONNECT UPDATE_IF) to overwrite existing records.

When accessing a sequential file randomly by relative record number, your program must provide the record number at symbolic offset RAB\$L_KBF and must specify a key length of 4 at symbolic offset RAB\$B_KSZ, in the RAB.

8.4.2 Processing Relative Files

The relative file organization permits greater program flexibility in performing record operations than the sequential organization. A program can read existing records from the file using sequential, random-by-relative-record-number (same as random-by-key), or random-by-RFA record access modes. You can write new records either sequentially or randomly, as long as the intended record location (cell) does not already contain a record. You can also delete records.

All record access modes for relative files allow you to establish the current-record position using the Find or Get service. After finding the record, VAX RMS permits you to delete the record from the relative file. After deleting the record, the empty cell becomes available for a new record. In addition, your program can update records anywhere in the file. If the records are variable length, the Update service can modify the record length up to the maximum size specified when the file was created.

When you insert a record into a relative file, the record is placed in a fixed cell within the file. A cell within a relative file can contain a record, can be vacant (never have contained a record), or can contain a deleted record.

The following sections discuss the sequential and random access modes for relative files.

8.4.2.1 Sequential Access

For relative files, the sequential access mode can be used to retrieve successive records in ascending record number. Vacant cells and cells that contain deleted records are skipped over automatically.

8.4.2.2 Random Access

You can directly read a record within a relative file by specifying the appropriate relative record number. If you attempt to read from a nonexistent cell — that is, a vacant cell or a cell containing a deleted record — VAX RMS returns an appropriate error message.

If you wish to position the record stream at a particular cell, regardless of whether or not it contains a record, you can do so by specifying the nonexistent-record option (FDL attribute CONNECT NONEXISTENT_RECORD) or by setting the RAB\$V_NXR bit in the RAB\$L_ROP field directly.

You can use two key record-processing options to directly access records in relative files: the *key-greater-than-or-equal* search option and the *key-greater-than* search option.

Note

For VAX/VMS Version 4.4, these options have been renamed at the VAX RMS level to reflect the VAX RMS expanded capability for searching indexed files in both ascending and descending sort order. The *key-greater-than-or-equal* search option is called the *equal-or-next* option and the *key-greater-than* option is called the *next* option. However, this applies only to indexed files — you are limited to searching relative files in ascending order by record numbers.

The *equal-or-next* option (FDL attribute CONNECT KEY_GREATER_EQUAL) directs VAX RMS to return a record having a record number equal to or greater than the specified record number. For example, if you specify record number 48, VAX RMS returns record number 48. If VAX RMS does not find record number 48, it returns the first record it encounters having a number greater than 48.

The *next* option (FDL attribute CONNECT KEY_GREATER_THAN) directs VAX RMS to return the record that has the next greater record number. For example, if you specify record number 48, VAX RMS returns record number 49, if record 49 exists.

You can also use random access mode to insert records into relative files. You can even overwrite cells that contain records by selecting the update-if option (FDL attribute CONNECT UPDATE_IF) or by directly setting the RAB\$V_UIF bit in the RAB\$L_ROP field.

To access a relative file randomly by record number, your program must specify the relative record number in the RAB at symbolic offset RAB\$L_KBF and a key length value of 4 at symbolic offset RAB\$B_KSZ.

8.4.3 Processing Indexed Files

Indexed files provide the most record-processing flexibility. Your program can read existing records from the file in sequential, random-by-RFA, or random-by-key record access mode. VAX RMS also allows you to write any number of new records into an indexed file if you do not violate a specified key characteristic, such as not allowing duplicate key values.

In random-by-key access mode, you can direct VAX RMS to use one of the search options in conjunction with one of four match options.

There are two search options:

- The *equal-or-next* search option (FDL attribute CONNECT KEY_GREATER_EQUAL) returns a record with a key value that equals the specified key value. If VAX RMS cannot find a record with the specified key value, it returns a record with a key value that meets the requirements of the specified sort order.
For example, assume an indexed file has four records having keys G, K, R and V, respectively. If the program wants to retrieve a record with key M and has specified ascending sort order, VAX RMS returns the record with key value R. Conversely, with descending sort order specified in this situation, VAX RMS returns the record with key value K.
- The *next* search option (FDL attribute CONNECT KEY_GREATER_THAN) returns a record having a key value that is not equal to the specified specified key value but meets the requirements of the specified sort order.

For example, assume an indexed file has five records having keys G, K, M, R and V, respectively. If the the program specifies the *next* search option with key value M and ascending sort order, VAX RMS returns the record with key value R. Conversely, with descending sort order specified in this situation, VAX RMS returns the record with key value K. In both cases, VAX RMS ignores the record with key value M.

You can perform a Find service, similar to the Get service, in sequential, random-by-RFA, or random-by-key record access mode. When finding records in random-by-key record access mode, your program can specify any one of the four types of key matches (exact, generic, approximate, generic /approximate) described previously in 2.1.1.2 and described later in 8.4.3.2.

For example, if the program specifies the *next* search option with a generic match on the three characters *RAM* using ascending sort order, VAX RMS will return records with key values *RAMA*, *RAMBO* and *RAMP* in that order. A record with key value *RAM* would not be returned. If descending sort order is specified, VAX RMS will return records with key values *RAMP*, *RAMBO* and *RAMA* in that order.

In addition to reading, writing, and finding a record, your program can delete or update any record in an indexed file if the operation does not violate specified key characteristics. For example, if the program specifies that key values cannot be changed, any update that attempts to change a key value is rejected.

The next section describes how indexed files are used with the sequential and random-by-key access modes.

8.4.3.1 Sequential Access

You can use sequential record access mode to retrieve successive records in an indexed file. VAX RMS retrieves the records in successive order by the specified sort order for a specific key of reference. The key of reference (for example, primary key, first alternate key, second alternate key, and so forth) is established through the use of one of the following services:

- Connect
- Rewind
- Find or Get using random access. (Note that a Get or Put service specifying random-by-RFA access always establishes the key of reference as the primary key.)

When the sequential access mode is used with the Put service to insert records into an indexed file, successive records must be in the specified sort order by primary key.

8.4.3.2 Random Access

One of the most useful features of indexed files is that you can randomly retrieve records by the record's key value. A key value and a key of reference (such as a primary key, first alternate key, and so forth) can be specified as input to the record-processing service. VAX RMS will then search the specified index to locate the record with the specified key value.

When reading records in random-by-key access mode, your program may specify one of four types of key matches:

- Exact key match
- Approximate key match
- Generic key match
- Approximate and generic key match

Exact match requires that the record's key value precisely match the key value specified by the program's Get service.

Approximate key match allows the program to select one of the following relationships between the key value of the retrieved record and the key value specified by the program:

- Equal to or greater than
- Greater than

The advantage of using an approximate key match is that your program can retrieve a record without knowing its precise key value. VAX RMS will use the approximations in your program to return the record with the key value nearest the specified value.

If you elect to use a generic key match, your program need provide only a specified number of leading characters in the key; for example, the first 5 bytes (characters) of a 10-byte string data-type key. VAX RMS uses this information to return the first record with a key value that begins with these characters and meets the specified sorting order requirement. This is useful when attempting to locate a record when only part of the key is known or for applications in which a series of records must be retrieved when only the initial portions of their key values are identical. Generic key match is available for string keys only.

When a generic key match is used with various approximate key match options, the results can vary, as shown in the following example. Consider using a key value of ABB to access records having key values of ABA, ABB and ABC, respectively.

- If the program elects to use the *equal-or-next* option with ascending sort order and a 3-character generic match, VAX RMS returns the record containing the key ABB.
- If the program uses the *next* option with ascending sort order and a 3-character generic match, VAX RMS returns the record with key value ABC.
- If the program uses the *equal-or-next* option with ascending sort order and a 2-character generic match, VAX RMS returns the record with key value ABA.

Now observe the effects of varying the key search option and the length of the generic string.

- If the program elects to use the *equal-or-next* option with ascending sort order and a 2-character generic match (AB), VAX RMS returns the record containing the key ABA.
- If the program uses the *next* option with descending sort order and a 3-character generic match, VAX RMS again returns the record with key value ABA.
- If the program uses the *next* option with descending sort order and a 2-character generic match (AB), VAX RMS returns a record- not-found condition because neither of the records has a key that begins with the letters AA.

Now consider an example of how to return all the records in a file with key values that match the generic string AB. You first specify the generic string value of AB (2-byte key) in random-by-key access mode. Then do the following:

- 1 Use the Get service (or the Find services) to access the first record
- 2 Change the record access mode to sequential
- 3 Access the next record
- 4 Compare the first two characters of the returned record's key with the first two characters of the specified key.
- 5 If the two key values are the same, process the record and return to step 3 above. If the two keys differ, do not process the record; instead, proceed to the next task (may require changing back to random-by-key access).

This procedure can be used to return all records that match a specified duplicate key for a key that allows duplicates. An alternative to checking the characters is to specify an ending key value and set the key-limit option when the record access mode is changed to sequential.

When accessing an indexed file randomly by key, the key value must reside in the area of memory identified by the control block offset `RAB$L_KBF`. When using string keys, you should specify the key length in the location identified by control block offset `RAB$B_KSZ`.

8.4.4 Access by Record's File Address

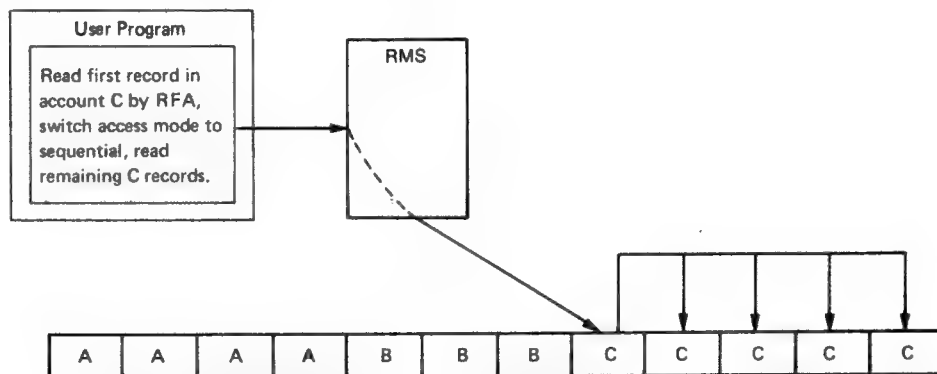
Random-by-RFA (record file address) access is supported for all disk files. Whenever VAX RMS successfully accesses a record, an internal representation of the record's location is returned in the 6-byte RAB field `RAB$W_RFA`. When a program wants to retrieve the record using random-by-RFA access, VAX RMS uses this internal data to retrieve the record.

One way to use RFA access is to establish a record position for subsequent sequential accesses. Consider a sequential file with variable-length records that can only be accessed randomly using RFA access. Assume the file consists of a list of transactions, sorted previously by account value. Because each account may have multiple transactions, each account value may have multiple records for it in the file. Instead of reading the entire file until it finds the first record for the desired account number, it uses a previously saved RFA value and random-by-RFA access to set the current-record position using a Find service at the first record of the desired account number. It can then switch to sequential record access and read all successive records for that account, until the account number changes or the end of the file is reached. Figure 8-1 shows how the file would be accessed for account C.

8.5 Block Input/Output

Block input/output (I/O) lets your program bypass the VAX RMS record-processing capabilities entirely. In this manner, your program can process a file as a virtually contiguous set of blocks.

Block I/O operations provide an intermediate step between VAX RMS operations and direct use of the Queue I/O Request system service. Using block I/O gives your program full control of the data in the individual blocks of a file while being able to take advantage of the VAX RMS capabilities for opening, closing, and extending a file.

Figure 8-1 Using RFA Access to Establish Record Position

ZK-753-82

In block I/O, a program reads or writes one or more blocks by specifying a starting virtual block number in the file and the length of the transfer. Regardless of the organization of the file, VAX RMS accesses the identified block or blocks.

Because VAX RMS files contain internal information meaningful only to VAX RMS itself, DIGITAL does not recommend that you modify an existing file using block I/O if the file is also to be accessed by VAX RMS record-level operations. (Block I/O does not update any internal record information.) The block I/O facility, however, does allow you to create your own file organizations. This file structure must be maintained through specialized user-written programs and procedures; VAX RMS cannot access these structures with its record access modes.

For more information about using block I/O, see the *VAX Record Management Services Reference Manual*.

8.6 Current Record Context

For each RAB connected to a FAB, VAX RMS maintains current context information about the record stream including the current-record position and the next-record position. Furthermore, the current context is different for the various VAX RMS services as shown in Table 8-2.

The current record context is internal to VAX RMS; you have no direct contact with it. However, you should know the context for each service in order to properly access records when you invoke a service.

Table 8-2 Record Access Stream Context

Record Operation	Record Access Mode	Current Record	Next Record
Connect	Does not apply	None	First record
Connect with RAB\$_ROP RAB\$_EOF bit set	Does not apply	None	End of file
Get, when last service was not a Find	Sequential	Old next record	New current record+1
Get, when last service was a Find	Sequential	Unchanged	Current record+1
Get	Random	New	New current record+1
Put, sequential file	Sequential	None	End of file
Put, relative file	Sequential	None	Next record position
Put, indexed file	Sequential	None	Undefined
Put	Random	None	Unchanged
Find	Sequential	Old next record	New current record+1
Find	Random	New	Unchanged
Update	Does not apply	None	Unchanged
Delete	Does not apply	None	Unchanged
Truncate	Does not apply	None	End of file

Table 8-2 (Cont.) Record Access Stream Context

Record Operation	Record Access Mode	Current Record	Next Record
Rewind	Does not apply	Unchanged	First record
Free	Does not apply	None	Unchanged
Release	Does not apply	None	Unchanged

Notes to Table 8-2:

- 1 Except for the Truncate service, VAX RMS establishes the current-record position before establishing the next-record position.
- 2 The notation "+1" indicates the next sequential record as determined by the file organization. For indexed files, the current key of reference is part of this determination.
- 3 The Connect service on an indexed file establishes the next record to be the first record in the index represented by the RAB key of reference (RAB\$B_KRF) field.
- 4 The Connect service leaves the next record as the end of file for a magnetic tape file opened for Put services (unless the FAB\$V_NEF option in the FAB\$L_FOP is set).

8.6.1 Current Record

For the Update, Delete, Release, and Truncate services, the current-record position reflects the location of the target record. The current-record position also facilitates sequential processing on disk devices for a stream.

Here is a list of situations where the current-record position is undefined:

- When a RAB is first connected to a FAB
- When a record operation is unsuccessful
- Following the successful execution of a VAX RMS service other than a Get or Find service.

When the current-record position is undefined, VAX RMS rejects any Update, Delete, Release, or Truncate service.

A Get service using sequential record access mode and immediately preceded by a Find service operates on the record specified by the current-record position. If the Find service does not lock the record (for relative and indexed files) and the current record is deleted, the Get service accesses the record at the next-record position.

Following successful execution of a Get or a Find service, the current-record position is set to the target record's RFA. VAX RMS also places the target record's address in the RFA field of the related RAB. The results are as follows:

- After initialization, the current-record position reflects the RFA of the record that was the object of the most recent successful Get or Find (unless failure occurs or a VAX RMS service other than Get or Find executes).
- Unless it is modified, the RAB\$W_RFA field always contains the address of the target current record. (If the operation fails, the RFA is undefined.)

Table 8-2 summarizes the effect that each successful record operation has on the context of the current record.

8.6.2 Next Record

VAX RMS uses the next-record position for doing sequential record access. For sequential record processing, the next-record position is the location of the target record for the next Find service (Get service where appropriate) or Put service. In a relative file, the target record is the record that occupies the next non-vacant cell.

The ability to look ahead significantly decreases access time for sequential processing. VAX RMS uses its internal knowledge of file organization and structures to determine the next-record position for each record service.

The Connect service initializes the next-record position to one of the following locations:

- The first record in a sequential file, or the first cell in a relative file.
- The first record in the collated sequence of the specified key of reference in an indexed file
- The end of a file on disk, if the RAB\$L_ROP field RAB\$V_EOF option is set

- The end of a write-accessed ANSI magnetic tape file, unless the FAB\$V_NEF option is set in the FAB\$L_FOP field

In any record access mode, the Get service establishes the next-record position as either the next record or the next record cell in the file. This is also true for the Find service in sequential access mode.

The Truncate service establishes the end of the file at the current-record position (effectively deleting the record at that location and all records following it) so you need only use Put services to extend the file. Note that you can only truncate sequential files.

In random access mode, the Find (or Get) service and the Put service do not affect the next-record position, unless these services are used to add a record with a primary key value or record number that lies between the corresponding values of the current record and the next record. When this occurs, the current-record position is changed to reflect the location of the added record—that is—records are added *after the current record*, not before the next record.

In sequential access mode, the Put service initializes the next-record position to the end of the file in a sequential file. In a relative file, the Put service initializes the next-record position to the next record or record cell. For sequential accesses to an indexed file, The Put service does not define the next-record position.

Regardless of access mode, the Delete, Update, Free, and Release services have no effect on the next-record position. For sequential and relative files, the Rewind service establishes the next-record position as the first record or record cell in the file, regardless of the access mode. In an indexed files, the Rewind service always establishes the next-record position as the location of the first record for the current key of reference.

Any unsuccessful record operation has no effect on the next record.

8.7 Synchronous and Asynchronous Operations

Your program can handle record operations on a file in one of two ways: synchronously or asynchronously. When operating synchronously, the program issuing the record-operation request regains control only when the request is completely satisfied. Most high-level languages support synchronous operation only. In asynchronous operations, the program can regain control before the request is completely satisfied. You can specify whether record operations will be performed synchronously or asynchronously for each record stream. File-level operations, such as opening a file, are always synchronous.

For instance, when reading a record from a file synchronously, the program regains control only after the record is passed to the program. In other words, the program is in a wait state until the record returns; no other processing for this program takes place during this read-and-return cycle. On the other hand, when reading a record asynchronously, the program might be able to regain control before the record is passed to the program. The program can thus use the time normally required for the record transfer between the file and memory to perform some other computations. Another record operation cannot be started on the same stream until the previous record operation is complete. However, record operations on other streams can be initiated.

Whether the program will actually regain control before the record operation is complete depends on several factors. For example, the required record may already reside in the I/O buffer, or the operating system may schedule another process, thus possibly allowing a necessary I/O operation to be completed before the original program is rescheduled.

One factor to consider in the use of asynchronous record operations is that you must include a separate completion routine or a VAX RMS wait request in the issuing program. This routine (or wait request) is required to determine when the record operation is complete because the results of the operation are not available and the next record operation for that stream cannot be initiated until the previous operation is completed.

8.7.1 Using Synchronous Operations

To declare a synchronous operation, you must clear the RAB\$V_ASY option in the RAB\$L_ROP field. Normally, you do not have to clear this option because it is already clear (by default). However, if the RAB\$V_ASY option had been set previously, then you must explicitly clear it.

Normally, you would not use success and error routines with synchronous operations. Instead, you would test the completion status code for an error and change the flow of the program accordingly. However, if you use these routines, they will be executed as asynchronous system traps (ASTs) before the VAX RMS service returns to your program (unless ASTs are disabled).

User-mode AST routines may be executed before the completion of a synchronous record operation, (see the *VAX Record Management Services Reference Manual*). If an AST routine attempts to perform operations on a record stream that is being called from a non-AST level, it must be prepared to handle stream-activity errors (RMS\$_RSA or RMS\$_BUSY).

8.7.2 Using Asynchronous Operations

To declare an asynchronous record operation, you must set the asynchronous (RAB\$V_ASY) option in the RAB\$L_ROP field. You can switch between synchronous and asynchronous operations during processing of a record stream by setting or clearing the RAB\$V_ASY option on a per-operation basis.

You can specify completion routines to be executed as ASTs if success or error conditions occur. Within such routines, you can issue additional operations, but they should also be asynchronous. If they are not, all other asynchronous requests currently active in your program cannot have their completion routines executed until the synchronous operation completes.

If an asynchronous operation is not yet complete at the time of return from a call to a VAX RMS service, the completion status field of the RAB will be 0, and a success status code of RMS\$_PENDING will be returned in Register 0. This status code indicates that the operation was initiated but is not yet complete. You must never modify the contents of a RAB when an operation is in progress.

If you issue a second record operation request for the same stream before a previous request has been completed, you will receive an RMS\$_RSA or RMS\$_BUSY error status code, indicating that the record stream is still active. This can also occur when an AST-level routine attempts to use an active record stream; the original I/O request may be synchronous or asynchronous. An additional error (RMS\$_BUSY) can be encountered by attempting an operation using the same record stream (RAB) from an error or success routine, when the main program is awaiting completion of the initial operation. In all cases, it is the programmer's responsibility to recognize this possibility and prevent the problem. Most problems can be prevented by issuing a Wait service; when the operation is complete, your program receives control at the point following the Wait service.

Note that the Connect operation may be performed asynchronously. If the RAB\$V_ASY option is set, a Wait service should follow the Connect service to synchronize with the completion of the Connect service. Another technique would be to perform the Connect service synchronously and set the RAB\$V_ASY option at run time, after the Connect service.

9

Run-Time Options

This chapter describes the way you specify run-time options and it summarizes the run-time options available to you when opening files, connecting record streams, processing records, and closing files. The run-time options that apply to record processing and to opening and closing a file can usually be preset by file-open and record stream connection values. Some options can be taken after you open a file and connect a record stream.

Note that run-time options discussed in previous sections are only summarized in this chapter. Most of the material in this chapter relates to options not previously described in this document.

9.1 Specifying Run-Time Options

This section describes the way you use the FDL Editor to specify run-time options that are available to your program through the FDL\$PARSE and FDL\$RELEASE routines. It also describes the use of language statements and VAX RMS to specify control block values.

You select VAX RMS options by setting appropriate values in VAX RMS control blocks within the data portion of your program. In many cases, you can select these values by using keywords available to you in the VAX language OPEN statement for your application, or by taking suitable default values. The values may be selected using keywords in your record and file description statements or they may be selected directly within the OPEN statement.

If your application is written in a VAX language that does not provide keywords for the various features, you can usually select the options using the File Definition Language (FDL).

Predefined FDL attributes can be supplied to your program at run time using the FDL\$PARSE routine. This routine also returns the address of the record access block (RAB) to let your program subsequently change RAB values. Some RAB options are not available in FDL and can be set only by directly accessing RAB fields and subfields at run time. To invoke options after record stream connection, your program must have direct access to VAX RMS control block fields using the address of the RAB and symbolic offsets into it.

9.1.1 Using the FDL Editor

You can use the FDL Editor to specify run-time attributes, such as adding a CONNECT attribute that will be used to set a control block value when the FDL\$PARSE and FDL\$RELEASE routines are called by your program. These attributes preset the values available for opening a file and connecting a record stream. An original FDL file created with the FDL Editor appears below.

```

IDENT  "19-JUL-1984 14:57:37  VAX-11 FDL Editor"
SYSTEM
SOURCE      VAX/VMS
FILE
ORGANIZATION  indexed
RECORD
CARRIAGE_CONTROL  carriage_return
FORMAT            variable
SIZE              0
AREA 0
ALLOCATION          8283
BEST_TRY_CONTIGUOUS  yes
BUCKET_SIZE        18
EXTENSION           2070
AREA 1
ALLOCATION          18
BEST_TRY_CONTIGUOUS  yes
BUCKET_SIZE        18
EXTENSION           18
KEY 0
CHANGES           no
DATA_AREA          0
DATA_FILL          100
DATA_KEY_COMPRESSION  yes
DATA_RECORD_COMPRESSION  yes
DUPLICATES         no
INDEX_AREA         1
INDEX_COMPRESSION   yes
INDEX_FILL         100
LEVEL1_INDEX_AREA  1
PROLOG             3
SEGO_LENGTH        9
SEGO_POSITION      0
TYPE               string

```

Because the FDL Editor does not include run-time attributes, you must add them to the FDL definition. You can specify run-time attributes by specifying the ACCESS, CONNECT and SHARING attributes. For example, if you want to add the CONNECT secondary attribute LOCK_ON_WRITE, you would use the EDIT/FDL ADD command. This is illustrated in the following example:

VAX-11 FDL Editor

- Add to insert one or more lines into the FDL definition
 Delete to remove one or more lines from the FDL definition
 Exit to leave the FDL Editor after creating the FDL file
 Help to obtain information about the FDL Editor
- 1 Invoke to initiate a script of related questions
 - Modify to change existing line(s) in the FDL definition
 - Quit to abort the FDL Editor with no FDL file creation
 - Set to specify FDL Editor characteristics
 - View to display the current FDL Definition
- 2 Main Editor Function (Keyword)[Help] : ADD
 Legal Primary Attributes
 ACCESS attributes set the run-time access mode of the file
 AREA x attributes define the characteristics of file area x
 CONNECT attributes set various VAX RMS run-time options
 DATE attributes set the data parameters of the file
 FILE attributes affect the entire VAX RMS data file
 JOURNAL attributes set the journaling parameters of the file
 KEY y attributes define the characteristics of key y
 RECORD attributes set the non-key aspects of each record
 SHARING attributes set the run-time sharing mode of the file
 SYSTEM attributes document operating system-specific items
 TITLE is the header line for the FDL file
 - 3 Enter Desired Primary (Keyword)[FILE] : CONNECT
 Legal CONNECT Secondary Attributes

ASYNCHRONOUS	yes/no	NOLOCK	yes/no
BLOCK_IO	yes/no	NONEXISTENT_RECORD	yes/no
BUCKET_CODE	number	READ_AHEAD	yes/no
CONTEXT	number	READ_REGARDLESS	yes/no
END_OF_FILE	yes/no	TIMEOUT_ENABLE	yes/no
FAST_DELETE	yes/no	TIMEOUT_PERIOD	number
FILL_BUCKETS	yes/no	TRUNCATE_ON_PUT	yes/no
KEY_GREATER_EQUAL	yes/no	TT_CANCEL_CONTROL_0	yes/no
KEY_GREATER_THAN	yes/no	TT_PROMPT	yes/no
KEY_LIMIT	yes/no	TT_PURGE_TYPE_AHEAD	yes/no
KEY_OF_REFERENCE	number	TT_READ_NOECHO	yes/no
LOCATE_MODE	yes/no	TT_READ_NOFILTER	yes/no
LOCK_ON_READ	yes/no	TT_UPCASE_INPUT	yes/no
LOCK_ON_WRITE	yes/no	UPDATE_IF	yes/no
MANUAL_UNLOCKING	yes/no	WAIT_FOR_RECORD	yes/no
MULTIBLOCK_COUNT	number	WRITE_BEHIND	yes/no
MULTIBUFFER_COUNT	number		
 - 4 Enter CONNECT Attribute (Keyword)[-] : LOCK_ON_WRITE
 - 5 CONNECT
 LOCK_ON_WRITE
 - 6 Enter value for this Secondary (Yes/No)[-] : YES
 Resulting Primary Section
 - 7 CONNECT
 LOCK_ON_WRITE yes
 - 8 Press RETURN to continue (~Z for Main Menu)

- ❶ This menu is the Main Editor Function menu. It displays the EDIT/FDL commands you can use.
- ❷ The ADD command displays the Legal Primary Attributes menu.
- ❸ The Legal Primary Attributes menu shows the primary attributes. You can either add a new primary attribute or add a secondary attribute to an existing primary attribute. Initially, the FILE primary attribute is the default.
- ❹ The selection of the CONNECT primary attribute displays the Legal CONNECT Secondary Attributes. You could similarly select the ACCESS, FILE, or SHARING options instead of the CONNECT primary attribute to display the Legal Secondary Attributes for the selected primary attribute.
- ❺ This menu shows all the CONNECT secondary attributes you can add to your FDL file.
- ❻ Select the proper CONNECT secondary (in this case, LOCK_ON_WRITE).
- ❼ EDIT/FDL verifies that you have selected the secondary.
- ❽ Enter the value that you want the secondary to have (for instance, *yes*).
- ❾ EDIT/FDL verifies the value for the secondary you have chosen.
- ❿ Return to the main menu. If you choose to add another secondary, you will notice that CONNECT is now the default.

Below is the completed FDL file containing the CONNECT primary attribute with the WRITE_BEHIND secondary attribute.

Run-Time Options

IDENT	"19-JUL-1984 14:57:37	VAX-11 FDL Editor"
SYSTEM	SOURCE	VAX/VMS
FILE	ORGANIZATION	indexed
RECORD	CARRIAGE_CONTROL	carriage_return
	FORMAT	variable
	SIZE	0
CONNECT	WRITE_BEHIND	yes
AREA 0	ALLOCATION	8283
	BEST_TRY_CONTIGUOUS	yes
	BUCKET_SIZE	18
	EXTENSION	2070
AREA 1	ALLOCATION	18
	BEST_TRY_CONTIGUOUS	yes
	BUCKET_SIZE	18
	EXTENSION	18
KEY 0	CHANGES	no
	DATA_AREA	0
	DATA_FILL	100
	DATA_KEY_COMPRESSION	yes
	DATA_RECORD_COMPRESSION	yes
	DUPLICATES	no
	INDEX_AREA	1
	INDEX_COMPRESSION	yes
	INDEX_FILL	100
	LEVEL1_INDEX_AREA	1
	PROLOG	3
	SEGO_LENGTH	9
	SEGO_POSITION	0
	TYPE	string

9.1.2 Using Language Statements and VAX RMS

Language statements such as OPEN may contain keywords, clauses, or other modifiers that correspond to the run-time attributes that are appropriate for opening files, connecting record stream, processing record and closing files. Some languages use system-defined procedures in place of keywords and clauses. Some languages allow you to call a user-supplied (USEROPEN or USERACTION) routine to set control block values before opening the file. For example, a user routine could be coded in VAX MACRO to take advantage of control block store macros (see Example 5-2 for an example of a VAX BASIC USEROPEN routine). Consult the appropriate language documentation for additional information.

With VAX MACRO, VAX RMS control block macros allow you to establish control block values at assembly time and, using the same control block, at run time. (The assembly-time macros are placed in a data section of the program; the run-time macros are placed in a code section of the program.) Using VAX MACRO, control blocks are allocated within the program space at assembly time and it may not be necessary to use the run-time macros because the program can move values to the control block fields using the VAX instruction set. Other languages, however, may not allocate the control blocks within program storage.

If your program has access to the starting location of the control block (a record access block, for instance), the VAX MACRO assembly-time control block macro or the corresponding symbol definition (DEF) macro provides your program with certain symbolic offsets (symbols) that can be used to locate and identify the various fields in the control block. Some VAX languages provide a means of making these symbols available to your program.

Refer to the *VAX Record Management Services Reference Manual* for additional information on using the control block macros and control block fields.

9.2 Options Related to Opening and Closing Files

Before your program can access the records in a file, it must open the file and connect a record stream. When it finishes processing records and no longer requires access to that file, your program should close the file.

The options available for opening files, connecting record streams and closing files include file access and file sharing options, file specification options, performance options, record access options, and options for:

- Adding records
- Acting on the file after it is closed (file disposition)
- Using indexed files
- Using magnetic tapes
- Performing nonstandard record processing
- Maintaining data reliability

9.2.1 File Access and Sharing Options

As described in Chapter 7, the program must declare the desired file-access and file-sharing values before opening an existing file or creating a new file and must specify record-locking and buffering strategies when the file is opened. These options are summarized below.

Option	Description
File access	Specifies the types of operations this process will perform: reading records, locating records, deleting records, adding new records, updating records, accessing blocks, and /or truncating the file. (For additional information, see Section 7.1.) You specify the file access values using the FDL ACCESS primary attribute or the VAX RMS FAB\$_FAC field.
File sharing	Specifies the types of operations this process will allow other file accessors to perform: reading records, locating records, deleting records, adding new records, and/or updating records. File sharing can also specify that this process will use multiple record streams (or ensure a read-only global buffer cache), that this process wishes to operate on the file without record interlocking, and/or disallow all other accessors from accessing the file. (For additional information, see Section 7.1.) You specify file sharing values using the SHARING primary attribute or the VAX RMS FAB\$_SHR field.
Record locking	Allows the user to elect to provide record locking for a shared file under user control. By default, VAX RMS will automatically lock records, depending on the file access and file sharing values specified. (For additional information, see Section 7.2.) You specify the record locking values using the CONNECT primary attribute or using the VAX RMS record-processing options (RAB\$_ROP) field ¹ .

¹Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

9.2.2 File Specifications

As described in Chapters 5 and 6, the program should specify the file specification of the file to be opened (or created) and can also specify default file specifications. The file specifications are summarized below.

File Specification	Description
Primary	Specifies the file specification to be used to locate the desired file(s). If any components of a file specification are omitted, VAX RMS applies defaults. You should always specify a primary file specification. FDL: FILE NAME VAX RMS: FAB\$_FNA and FAB\$_FNS
Default	Specifies the default file specification to be used to fill any missing components not provided by the primary file specification. After applying these defaults, if any components are still missing, additional defaults are applied. FDL: FILE DEFAULT_NAME VAX RMS: FAB\$_DNA and FAB\$_DNS
Related	Specifies a related file specification that is used to provide additional defaults, if a secondary (related) file is being used. After applying the defaults in the related file specification (if any), if the device or directory components are still missing, an additional default is provided if the component is missing using the process-default device (SYS\$DISK) and current process-default directory (which VAX RMS maintains). FDL: None. VAX RMS: FAB\$_NAM and NAM\$_RLF

9.2.3 File Performance Options

A number of run-time file options that open files and connect record streams can collectively improve application performance. Such options include the buffering options discussed in Chapter 7.

Two run-time performance options not discussed previously are particularly important when adding records to a file: extension size and window size.

9.2.3.1 Extension Size

If you intend to add records to the file, you should specify a reasonable default extension size to reduce the number of times that the file will be extended.

You should always use the Edit/FDL Utility to calculate the correct extension size. EDIT/FDL uses your responses to script questions to assign an optimum value for the FDL attribute FILE EXTENSION. With multiple area files, EDIT/FDL also assigns optimum values to the AREA EXTENSION attributes.

If you do not specify an extension size, VAX RMS computes the size; however, this size may not be optimum.

If you decide to create an FDL file for defining an indexed file without using EDIT/FDL, you can approximate the value of the EXTENSION attributes. You do this by multiplying number of records per bucket by the number of records that you intend to add to the file during a given period of time.

To see what the current default extension size is, use the DCL command SHOW RMS_DEFAULT. To set the default buffer count, use the DCL command SET RMS_DEFAULT/EXTEND_QUANTITY=*n*, where *n* is the number of blocks per extension. The corresponding VAX RMS field is FAB\$B_DEQ.

9.2.3.2 Window Size

If the file is extended repeatedly, the extensions may be scattered on the disk. Each extension is called an *extent*—a pointer to each extent resides in the file header. For retrieval purposes, the pointers are gathered together in a structure called a *window*. The default window size is 7 pointers, but you can establish the window size to contain as many as 127 pointers. You can also set the window size to -1, which makes a window that is just large enough to map the entire file.

When you access an extent whose pointer is not in the current window, the system has to read the file header and fetch the appropriate window. This is called a *window turn*, and it requires an I/O operation.

Window size is a run-time option. Many of the high-level languages have a clause that sets window size during the opening of the file. You can also set the window size (FAB\$B_RTV field) at run time with a VAX MACRO subroutine or specify the FDL attribute FILE WINDOW_SIZE.

You can increase the default window size for a specific disk volume by using the DCL commands MOUNT and INITIALIZE. However, using additional window pointers increases system overhead. The window size is charged to your buffered I/O byte count quota and indiscriminate use of large windows may result in exceeding this quota to be exceeded or may exhaust the system's nonpaged dynamic memory.

You can use the Backup Utility (BACKUP) to avoid having too many extents. When you restore a file, BACKUP tries to write the file in one section of the disk. Although BACKUP does not necessarily create a contiguous copy of the file, it does reduce the number of extents. If you are regularly backing up the file, the number of extents is probably reasonable. For more information about BACKUP, see the *VAX/VMS Backup Utility Reference Manual*.

Where disk space is available, you can reduce the number of extents by creating a new, contiguous version of the file using either the Convert Utility (CONVERT) or the DCL command COPY/CONTIGUOUS. If neither of these conditions apply, a larger window size is the only option to use. For file maintenance information, see Chapter 10.

9.2.3.3 Summary of Performance Options

The following list summarizes the run-time open and connect options that may affect performance.

Option	Description
Asynchronous record processing ¹	Specifies that record I/O for this record stream will be done asynchronously. See Section 8.3 for more information. FDL: CONNECT ASYNCHRONOUS VAX RMS: RAB\$_ROP RAB\$_VASY
Deferred write ¹	Allows records to be accumulated in a buffer and written only when the buffer is needed or when the file is closed. For use by all except non-shared sequential files. See Chapter 3. FDL: FILE DEFERRED_WRITE VAX RMS: FAB\$_FOP FAB\$_DFW

¹Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

Option	Description
Default extension quantity	Specifies the number of blocks to be allocated to a file when more space is needed. FDL: FILE EXTENSION VAX RMS: FAB\$W_DEQ
Fast delete ¹	Postpones certain internal operations associated with deleting indexed file records until the record is accessed again. This allows records to be deleted rapidly, but may affect the performance of subsequent accessors reading the file. FDL: CONNECT FAST_DELETE VAX RMS: RAB\$L_ROP RAB\$V_FDL
Global buffer count	Specifies whether global buffers will be used and the number to be used if the record stream is the first to connect to the file. See Section 7.3. FDL: CONNECT GLOBAL_BUFFER_COUNT VAX RMS: FAB\$W_GBC
Locate mode ¹	Allows the use of locate mode, not move mode, when reading records. See Section 7.3. FDL: CONNECT LOCATE_MODE VAX RMS: RAB\$L_ROP RAB\$V_LOC
Multiblock count	Allows multiple blocks to be transferred into memory during a single I/O operation; for sequential files only. See Section 7.3 and Chapter 3. FDL: CONNECT MULTIBLOCK_COUNT VAX RMS: RAB\$B_MBC
Number of buffers	Enables the use of multiple buffers; used with indexed and relative files for the buffer cache and sequential files for the read-ahead and write-behind options. See Section 7.3. FDL: CONNECT MULTIBUFFER_COUNT VAX RMS: RAB\$B_MBF
Read-ahead ¹	Alternates buffer use between two buffers; for sequential files only. See Chapter 2. FDL: CONNECT READ_AHEAD VAX RMS: RAB\$L_ROP RAB\$V_RAH

¹Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

Option	Description
Retrieval window size	Specifies the number of entries in memory for retrieval windows, which corresponds to the number of extents for a file. FDL: FILE WINDOW_SIZE VAX RMS: FAB\$_RTV
Sequential access only	Indicates that the file will be accessed sequentially only; for sequential files only. FDL: FILE SEQUENTIAL_ONLY VAX RMS: FAB\$_FOP FAB\$_SQO
Write-behind ¹	Alternates buffer use between two buffers; for sequential files only. See Chapter 2. FDL: CONNECT WRITE_BEHIND VAX RMS: RAB\$_ROP RAB\$_WBH

¹Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

9.2.4 Record Access Options

You can specify the record access for a record stream as sequential, random by key or record number, or random by record's file address. (See Section 8.1 for more information.) The record access chosen can be changed for each record processing operation. These options can be set using the VAX RMS RAB\$_RAC field, values RAB\$_SEQ, RAB\$_KEY, and RAB\$_RFA.

9.2.5 Options for Adding Records

When adding records to a file, several file open and connection options, which are listed below, should be considered.

Option	Description
Default extension quantity ¹	See Section 9.2.3
Deferred write ¹	See Section 9.2.3

¹Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

Option	Description
End-of-file	After the record stream is connected, the record context is positioned to the end of the file. FDL: CONNECT END_OF_FILE VAX RMS: RAB\$L_ROP RAB\$V_EOF
Retrieval window size ¹	See Section 9.2.3
Revision data	The revision date-time and the revision number can be specified to be a value other than the actual revision date-time and revision number when the file is closed. These options must be set between the time the file is opened and the file is closed and thus cannot be set using FDL. FDL: Does not apply. VAX RMS: Revision Date and Time XAB
Truncate on Put ¹	When using sequential record access for sequential files only, the record to be written will be the last record in the file and VAX RMS should truncate the file just beyond that record. FDL: CONNECT TRUNCATE_ON_PUT VAX RMS: RAB\$L_ROP RAB\$V_TPT
Update-if ¹	If you set this option and your program tries to replace an existing record while adding records randomly to a file, VAX RMS will modify the existing record instead of replacing it. When using this option for indexed files, note that the file must not allow duplicates for the primary key. This option should be used with care with a shared file (see Section 8.4 for more information). FDL: CONNECT UPDATE_IF VAX RMS: RAB\$L_ROP RAB\$V_UIF
Write-behind ¹	See Section 9.2.3

¹Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

9.2.6 Options for Data Reliability

The run-time file open options that apply to data reliability are listed in the following table:

Option	Description
Read-check	Specifies that transfers from disk volumes are to be checked by a read-compare operation, which effectively doubles the amount of disk I/O performed. Not available for all devices (see the <i>VAX Record Management Services Reference Manual</i>). FDL: FILE READ_CHECK VAX RMS: FAB\$L_FOP FAB\$V_RCK
Write-check	Specifies that transfers to disk volumes are to be checked by a read-compare operation, which effectively doubles the amount of disk I/O performed. Not available for all devices (see the <i>VAX Record Management Services Reference Manual</i>). FDL: FILE WRITE_CHECK VAX RMS: FAB\$L_FOP FAB\$V_WCK

9.2.7 Options for File Disposition

Certain options available when opening, connecting, or closing a file allow you to specify that the file is to be deleted when it is closed, spooled for printing, or submitted as a command procedure.

The run-time file open options that apply to file disposition are listed below. These options are effective if set before the file is closed (such as when the file is opened or between the time the file is opened and closed).

Option	Description
Delete on close	Deletes the file when it is closed. FDL: CONNECT DELETE_ON_CLOSE VAX RMS: FAB\$L_FOP FAB\$V_DLT
Submit command file	Submits the file as a batch command procedure to SYSS\$BATCH when the file is closed; for sequential files only. FDL: FILE SUBMIT_ON_CLOSE VAX RMS: FAB\$L_FOP FAB\$V_SCF
Spool on close	Prints the file on SYSS\$PRINT when the file is closed; for sequential files only. FDL: FILE PRINT_ON_CLOSE VAX RMS: FAB\$L_FOP FAB\$V_SPL

9.2.8 Options for Indexed Files

The list of run-time options that apply to indexed file processing are listed below. Refer to Section 8.1.4 for more information.

Option	Description
Fast delete ¹	<p>This option lets you postpone certain internal operations associated with deleting indexed file records until the record is next accessed. This allows records to be deleted rapidly, but may affect the performance of subsequent accessors reading the file.</p> <p>FDL: CONNECT FAST_DELETE VAX RMS: RAB\$L_ROP RAB\$V_FDL</p>
Key equal or next ¹	<p>If you take this option when locating or reading records, VAX RMS returns the first record with a key value equal to the specified key. If VAX RMS does not find a record with an equal key value, it returns the record with the next higher key value when ascending sort order is specified. When descending sort order is specified, VAX RMS returns the next record with the next lower key value.</p> <p>FDL: CONNECT KEY_GREATER_EQUAL VAX RMS: RAB\$L_ROP RAB\$V_EQNXT</p>
Next key ¹	<p>If you take this option when locating or reading records, VAX RMS returns the record with the next higher key value when you specify ascending sort order. When you specify descending sort order, VAX RMS returns the next record with the next lower key value. If you do not specify either this option or the key-equal-or-next option, VAX RMS tries for a key-equal match.</p> <p>FDL: CONNECT KEY_GREATER_THAN VAX RMS: RAB\$L_ROP RAB\$V_NXT</p>
Key of reference	<p>When you are processing an indexed file with multiple keys, this option lets you specify which key will be used for the current record stream.</p> <p>FDL: CONNECT KEY_OF_REFERENCE VAX RMS: RAB\$B_KRF</p>

¹Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

Option	Description
Key buffer ¹	<p>If you take this option when locating or reading records randomly, the specified key buffer must contain the selected record's key.</p> <p>FDL: None. VAX RMS: RAB\$L_KBF</p>
Key size ¹	<p>If you take this option when locating or reading records with a string key type, you can specify that only a portion of the key be used to locate the selected record.</p> <p>FDL: None. VAX RMS: RAB\$B_KSZ</p>
Limit key ¹	<p>If you take this option when locating or reading records sequentially (not randomly), VAX RMS returns an alternate success status when it encounters a record with a key value that exceeds the specified limit value.</p> <p>FDL: CONNECT KEY_LIMIT VAX RMS: RAB\$L_ROP RAB\$V_LIM</p>
Load buckets ¹	<p>If you take this option when adding records to an index file, VAX RMS will use the fill factor specified when the file was created before a bucket split occurs. If this option is not specified, buckets will be filled completely before a bucket split occurs.</p> <p>FDL: CONNECT FILL_BUCKETS VAX RMS: RAB\$L_ROP RAB\$V_LOA</p>

¹Indicates an option that can be specified for each record-processing operation. For more information, see Section 9.3.

9.2.9 Options for Magnetic Tape Processing

The run-time file open and close options that apply to magnetic tape processing are listed below.

Option	Description
Not end-of-file	If you take this option when adding records (ACCESS PUT), the record position should not be at the end of the file. FDL: FILE MT_NOT_EOF VAX RMS: FAB\$L_FOP FAB\$V_NEF
Current position	If you take this option when creating a file, the tape is positioned to the location immediately following the most recently closed file. FDL: FILE MT_CURRENT_POSITION VAX RMS: FAB\$L_FOP FAB\$V_POS
Rewind on Open	If you take this option, VAX RMS rewinds the tape before it opens or creates the file. Overrides the current-position option. FDL: FILE MT_OPEN_REWIND VAX RMS: FAB\$L_FOP FAB\$V_RWO
Rewind on Close	If you take this option, VAX RMS rewinds the tape volume before it closes the file. FDL: FILE MT_CLOSE_REWIND VAX RMS: FAB\$L_FOP FAB\$V_RWC

9.2.10 Options for Nonstandard File Processing

The run-time file open options that apply to nonstandard file processing are listed below.

Option	Description
Non-file-structured	The volume is to be processed in a non-file-structured manner, allowing use of volumes created on non-DIGITAL systems. FDL: FILE NON_FILE_STRUCTURED VAX RMS: FAB\$L_FOP FAB\$V_NFS
User file open	VAX RMS will only open the file; Queue I/O Request system service calls will be used to access the contents of the file, whose channel number is returned in the FAB\$L_STV field. FDL: FILE USER_FILE_OPEN VAX RMS: FAB\$L_FOP FAB\$V_UFO

9.3 Summary of Record Operation Options

This section briefly describes the options associated with the record retrieval (Find and Get) services, the record insertion (Put) service, the record modification (Update) service, and the record deletion (Delete) service.

9.3.1 Record Retrieval Options

The Find and Get services (or the equivalent VAX language statements) can be used to locate and retrieve a record.

The options associated with the Find and Get services are summarized below. These options can be set for each Find or Get service if the program can access the appropriate RAB control block fields. The RAB control block fields are preset by connect-time values or defaults and as a result of previous VAX RMS service calls.

Option	Description
Asynchronous record processing	Specifies that record I/O for this record stream will be done asynchronously. FDL: CONNECT ASYNCHRONOUS VAX RMS: RAB\$L_ROP RAB\$V_ASY
Do not lock record	Indicates that VAX RMS will not lock the record for the following record operations. FDL: CONNECT NOLOCK VAX RMS: RAB\$L_ROP RAB\$V_NLK
Key buffer	When locating/reading records randomly, the specified key buffer must contain the desired record's key. FDL: None. VAX RMS: RAB\$L_KBF
Key equal or next	When locating or reading records, VAX RMS returns the first record with a key value equal to the specified key. If VAX RMS does not find a record with an equal key value, it returns the record with the next higher key value when you specify ascending sort order. When you specify descending sort order, VAX RMS returns the record with the next lower key value. FDL: CONNECT KEY_GREATER_EQUAL VAX RMS: RAB\$L_ROP RAB\$V_EQNXT

Option	Description
Next key	<p>When locating or reading records, VAX RMS returns the record with the next higher key value when you specify ascending sort order. When you specify descending sort order, VAX RMS returns the record with the next lower key value.</p> <p>FDL: CONNECT KEY_GREATER_THAN VAX RMS: RAB\$L_ROP RAB\$V_NXT</p>
Key of reference	<p>For indexed files with multiple keys, specifies which key will be used for this record stream.</p> <p>FDL: CONNECT KEY_OF_REFERENCE VAX RMS: RAB\$B_KRF</p>
Key size	<p>When locating/reading records using a string key type, you can specify that only a portion of the key be used to locate the desired record(s).</p> <p>FDL: None. VAX RMS: RAB\$B_KSZ</p>
Limit key	<p>When locating/reading records in an indexed file sequentially (not randomly), VAX RMS is to return an alternate success status when the key in the record exceeds the specified key.</p> <p>FDL: CONNECT KEY_LIMIT VAX RMS: RAB\$L_ROP RAB\$V_LIM</p>
Locate mode	<p>Specifies that locate mode, not move mode, will be used. Applies to the Get service only.</p> <p>FDL: CONNECT LOCATE_MODE VAX RMS: RAB\$L_ROP RAB\$V_LOC</p>
Lock nonexistent record	<p>Indicates that VAX RMS is to lock the record position at the location of the following record operation, regardless of whether a record exists at that location. Applies only to relative files.</p> <p>FDL: CONNECT NONEXISTENT_ VAX RMS: RECORD RAB\$L_ROP RAB\$V_NXR</p>
Lock for read	<p>Locks record for reading and allow other readers (but no writers).</p> <p>FDL: CONNECT LOCK_ON_READ VAX RMS: RAB\$L_ROP RAB\$V_REA</p>

Option	Description
Lock for write	Locks record for writing and allows other readers (but no writers). FDL: CONNECT LOCK_ON_WRITE VAX RMS: RAB\$L_ROP RAB\$V_RLK
Manual locking	Indicates that the user will control record locking and unlocking manually. FDL: CONNECT MANUAL_LOCKING VAX RMS: RAB\$L_ROP RAB\$V_ULK
Read-ahead	Improves performance at the expense of additional memory for I/O buffers. For sequential access to sequential files only. FDL: CONNECT READ_AHEAD VAX RMS: RAB\$L_ROP RAB\$V_RAH
Read regardless	Reads the specified record regardless of whether it is locked by another user. FDL: CONNECT READ_REGARDLESS VAX RMS: RAB\$L_ROP RAB\$V_RRL
Record access	Specifies the way records will be accessed, sequentially, randomly by key (indexed files) or by record number (relative files), or randomly by record's file address. FDL: None. VAX RMS: RAB\$B_RAC values RAB\$C_SEQ, RAB\$C_KEY, RAB\$C_RFA
Record's file address	Specifies the record's file address (RFA) of the desired record when records are being accessed randomly by RFA (RAB\$B_RAC contains RAB\$C_RFA). This value is also returned by Find and Get services regardless of the type record access used. FDL: None. VAX RMS: RAB\$W_RFA
Record header buffer	Defines the length of the fixed portion of variable with fixed control (VFC) records. Applies to the Get service only. FDL: None. VAX RMS: RAB\$L_RHB

Option	Description
Timeout period	<p>If the wait-if-locked option is specified, this option may be specified to specify a timeout period after which an error is returned. The number of seconds is specified by the CONNECT TIMEOUT_PERIOD or RAB\$B_TMO field to eliminate a potential deadlock.</p> <p>FDL: CONNECT TIMEOUT_PERIOD</p> <p>VAX RMS: RAB\$L_ROP RAB\$V_TMO and RAB\$B_TMO</p>
User buffer address	<p>Specifies the address of the user buffer that will receive the read record. Applies to the Get service only.</p> <p>FDL: None.</p> <p>VAX RMS: RAB\$L_UBF</p>
User buffer size	<p>Specifies the maximum length of the user record buffer. Applies to the Get service only.</p> <p>FDL: None.</p> <p>VAX RMS: RAB\$L_USZ</p>
Wait if locked	<p>If the record is locked, wait until it is available; also allows use of the timeout-period option.</p> <p>FDL: CONNECT WAIT_FOR_RECORD</p> <p>VAX RMS: RAB\$L_ROP RAB\$V_WAT</p>

9.3.2 Record Insertion Options

The Put service (or equivalent VAX language statement) adds a record to the file.

The options associated with the Put service are summarized below. These options can be set for each Put service if the program can access the appropriate RAB control block fields. The RAB control block fields are preset by connect-time values or defaults and as a result of previous VAX RMS service calls.

Run-Time Options

Option	Description
Asynchronous record processing	<p>Specifies that record I/O for this record stream will be done asynchronously.</p> <p>FDL: CONNECT ASYNCHRONOUS VAX RMS: RAB\$L_ROP RAB\$V_ASY</p>
Deferred write	<p>Allows records to be accumulated in a buffer and only written when the buffer is needed or when the file is closed. For relative and indexed files only.</p> <p>FDL: FILE DEFERRED_WRITE VAX RMS: FAB\$L_FOP FAB\$V_DFW</p>
Key buffer	<p>When adding records randomly to a relative file, the specified key buffer must contain the desired record's relative record number.</p> <p>FDL: None. VAX RMS: RAB\$L_KBF</p>
Key size	<p>When adding records to a relative file using random record access, this field must specify a value of 4 (although this is the default provided by VAX RMS).</p> <p>FDL: None. VAX RMS: RAB\$B_KSZ</p>
Load buckets	<p>When adding records, the buckets will be filled only to the fill size specified when the file was created before a bucket split occurs. If this option is not specified, buckets will be filled completely before a bucket split occurs.</p> <p>FDL: CONNECT FILL_BUCKETS VAX RMS: RAB\$L_ROP RAB\$V_LOA</p>
Read allowed	<p>Allows the locked record being written to be read.</p> <p>FDL: CONNECT LOCK_ON_WRITE VAX RMS: RAB\$L_ROP RAB\$V_RLK</p>
Record access	<p>Specifies the way records will be added, sequentially according to ascending key value or relative record number, randomly by key (indexed files) or by record number (relative files), or randomly by record's file address.</p> <p>FDL: None. VAX RMS: RAB\$B_RAC values RAB\$C_SEQ, RAB\$C_KEY, RAB\$C_RFA</p>

Run-Time Options

Option	Description
Record header buffer	Defines the length of the fixed portion of variable with fixed control (VFC) records. FDL: None. VAX RMS: RAB\$L_RHB
Record buffer address	Specifies the address of the record buffer that contains the record to be written. FDL: None. VAX RMS: RAB\$L_RBF
Record buffer size	Specifies the size of the record contained in the record buffer to be written. FDL: None. VAX RMS: RAB\$L_RBZ
Timeout period	If the wait-if-locked option is specified, this option may be specified to specify a timeout period after which an error is returned. The number of seconds is specified by the CONNECT TIMEOUT_PERIOD or RAB\$B_TMO field to eliminate a potential deadlock. FDL: CONNECT TIMEOUT_PERIOD VAX RMS: RAB\$L_ROP RAB\$V_TMO and RAB\$B_TMO
Truncate on Put	Specifies that the file is to be truncated at the position of the record being added. Requires sequential record access and only applies to sequential files. FDL: CONNECT TRUNCATE_ON_PUT VAX RMS: RAB\$L_ROP RAB\$V_TPT

Option	Description
Update-if	<p>Turns the Put service into an update operation if the record already exists in the file. Care must be taken when using this option with shared files and automatic record locking (see Section 8.4). When using this option with indexed files, note that the file must not allow duplicates for the primary key. This option can only be used when random record access has been specified.</p> <p>FDL: CONNECT UPDATE_IF VAX RMS: RAB\$L_ROP RAB\$V_UIF</p>
Write-behind	<p>Improves performance at the expense of additional memory for I/O buffers. For sequential access to sequential files only.</p> <p>FDL: CONNECT WRITE_BEHIND VAX RMS: RAB\$L_ROP RAB\$V_WBH</p>

9.3.3 Record Update Options

The Update service (or equivalent VAX language statement) modifies an existing record in a file. Your program must first locate the appropriate record position and optionally retrieve the record itself, by calling the Find or Get service (or equivalent VAX language statement).

The options associated with the Update service are summarized below. These options can be set for each Update service if the program can access the appropriate RAB control block fields. The RAB control block fields are preset by connect-time values or defaults and as a result of previous VAX RMS service calls.

Option	Description
Asynchronous record processing	<p>Specifies that record I/O for this record stream will be done asynchronously.</p> <p>FDL: CONNECT ASYNCHRONOUS VAX RMS: RAB\$L_ROP RAB\$V_ASY</p>
Record header buffer	<p>Defines the length of the fixed portion of variable with fixed control (VFC) records.</p> <p>FDL: None. VAX RMS: RAB\$L_RHB</p>

Option	Description
Record buffer address	Specifies the address of the record buffer that contains the record to be written. FDL: None. VAX RMS: RAB\$L_RBF
Record buffer size	Specifies the size of the records contained in the record buffer to be written. FDL: None. VAX RMS: RAB\$L_RBZ

9.3.4 Record Deletion Options

The Delete service (or equivalent VAX language statement) removes a record from the file. You cannot use this service for sequential files, however, a sequential file can be truncated using the Truncate service. Like the Update service, the Delete must be preceded by a Find or Get service to establish the current record position.

The options associated with the Delete service are summarized below. These options can be set for each Delete service if the program can access the appropriate RAB control block fields. The RAB control block fields are preset by connect-time values or defaults and as a result of previous VAX RMS service calls.

Option	Description
Asynchronous record processing	Specifies that record I/O for this record service will be done asynchronously. FDL: CONNECT ASYNCHRONOUS VAX RMS: RAB\$L_ROP RAB\$V_ASY
Fast delete	Specifies that the record to be deleted will be flagged as deleted but parts of any alternate index key path will not be completely erased until a subsequent access using the alternate key occurs. This makes deleting the record occur more quickly, but will require additional access time for a subsequent Find or Get service. FDL: CONNECT FAST_DELETE VAX RMS: RAB\$L_ROP RAB\$V_FDL

9.4 Run-Time Example

Example 9-1 shows how to invoke the FDL\$PARSE and FDL\$RELEASE routines to use the predefined control block values set by an EDIT/FDL editing session.

Example 9-1 Using FDL\$PARSE and FDL\$RELEASE

```

;
; This program calls the FDL utility routines FDL$PARSE and
; FDL$RELEASE. First, FDL$PARSE parses the FDL specification
; PART.FDL. Then the data file named in PART.FDL is accessed
; using the primary key. Last, the control blocks allocated
; by FDL$PARSE are released by FDL$RELEASE.
;
; .TITLE FDLXAM
;
; .PSECT DATA,WRT,NOEXE
;
MY_FAB: .LONG 0
MY_RAB: .LONG 0
FDL_FILE: .ASCID /PART.FDL/ ; Declare FDL file
REC_SIZE=80
LF=10
REC_RESULT: .LONG REC_SIZE
; .ADDRESS REC_BUFFER
REC_BUFFER: .BLKB REC_SIZE
HEADING: .ASCID /ID PART SUPPLIER COLOR /<LF>
;
; .PSECT CODE
;
; Declare the external routines
;
; .EXTRN FDL$PARSE, -
; FDL$RELEASE
;
; .ENTRY FDLXAM,"M<>" ; Set up entry mask
; PUSHAL MY_RAB ; Get set up for call with
; PUSHAL MY_FAB ; addresses to receive the
; PUSHAL FDL_FILE ; FAB and RAB allocated by
; CALLS #3,G~FDL$PARSE ; FDL$PARSE
; BLBS RO,KEYO ; Branch on success
; BRW ERROR ; Signal error
;

```

(Continued on next page)

Example 9-1 (Cont.) Using FDL\$PARSE and FDL\$RELEASE

```

KEYO:      MOVL    MY_FAB,R10          ; Move address of FAB to R10
           MOVL    MY_RAB,R9          ; Move address of RAB to R9
           MOVL    $REC_SIZE,RAB$W_USZ(R9)
           MOVAB   REC_BUFFER,RAB$L_UBF(R9)
           $OPEN   FAB=(R10)          ; Open the file
           BLBC    RO,F_ERROR
           $CONNECT RAB=(R9)          ; Connect to the RAB
           BLBC    RO,R_ERROR
           PUSHAQ  HEADING            ; Display the heading
           CALLS   #1,G~LIB$PUT_OUTPUT
           BLBC    RO,ERROR
           BRB     GET_REC            ; Skip error handling

;
F_ERROR:   BRW     FAB_ERROR
R_ERROR:   BRW     RAB_ERROR
;
GET_REC:   $GET    RAB=(R9)            ; Get a record
           CMPL    $RMS$_EOF,RO        ; If not end of file,
           BEQLU   CLEAN              ; continue
           BLBC    RO,R_ERROR
           MOVZWL   RAB$W_RSZ(R9),REC_RESULT ; Move a record into
           PUSHAL   REC_RESULT          ; the buffer
           CALLS   #1,G~LIB$PUT_OUTPUT  ; Display the record
           BLBC    RO,ERROR
           BRB     GET_REC            ; Get another record

CLEAN:     $CLOSE   FAB=(R10)          ; Close the FAB
           BLBC    RO,FAB_ERROR
           PUSHAL   MY_RAB             ; Push RAB address on stack
           PUSHAL   MY_FAB             ; Push FAB address on stack
           CALLS   #2,G~FDL$RELEASE    ; Release the control blocks
           BLBC    RO,ERROR
           BRB     FINI               ; Successful completion

;
FAB_ERROR: PUSHL    FAB$L_STV(R10)      ; Signal file error
           PUSHL    FAB$L_STS(R10)
           BRB     RMS_ERR
;
;
ERROR:     PUSHL    RO                  ; Signal error
           CALLS   #1,G~LIB$SIGNAL
           $CLOSE   FAB=(R10)
           BRW     FINI               ; End program

```

(Continued on next page)

Example 9-1 (Cont.) Using FDL\$PARSE and FDL\$RELEASE

```
;
RAB_ERROR:      PUSHL   RAB$L_STV(R9)      ; Signal record error
                PUSHL   RAB$L_STS(R9)
;
RMS_ERR:        CALLS   #2,G~LIB$SIGNAL
;
FINI:           RET
                .END   FDLEXAM
```

10 Maintaining Files

Designing and creating your files and defining their records is only the first step in the life cycle of your file. You must also consider maintaining the file.

This chapter describes file maintenance with the emphasis on file tuning.

Section 10.1 describes how you can use the Analyze/RMS_File Utility to view the characteristics of a file. Section 10.3 explains how to use the Edit/FDL Utility, particularly with Analyze/RMS_File, to optimize and redesign file characteristics. Section 10.4 describes how to make a file contiguous. Section 10.5 explains how to reorganize a file, and Section 10.6 describes how to make archive copies of a file.

10.1 Viewing File Characteristics

The Analyze/RMS_File Utility (ANALYZE/RMS_FILE) allows you to inspect and analyze the internal structure of a VAX RMS file.

ANALYZE/RMS_FILE can check a file's structure for errors and can generate a statistical or summary report. A summary report is identical to a statistical report except that no checking is done. For more information on producing a summary report, see the description of the Analyze/RMS_File Utility in the *VAX/VMS Analyze/RMS_File Utility Reference Manual*.

You can also inspect and analyze your file using the Analyze/RMS_File Utility interactively. The analysis can show whether or not the file is properly designed for its application and can point out ways to improve the file design.

In addition, you can use ANALYZE/RMS_FILE to FDL files from data files. You can then use these FDL files with the Create/FDL Utility (CREATE/FDL), the Convert Utility (CONVERT), and the Edit/FDL Utility (EDIT/FDL). FDL files created with ANALYZE/RMS_FILE contain special analysis sections for each area and key, which are called ANALYSIS_OF_AREA and ANALYSIS_OF_KEY. The Edit/FDL Utility uses these sections in the Optimize script to tune the file's structure.

10.1.1 Performing an Error Check

To check a file's structure for errors, enter a command of this form at the DCL command level:

ANALYZE/RMS_FILE/CHECK file-spec

By default with a command of this format, the Check report is displayed on the terminal (SYS\$OUTPUT).

If you receive any error messages, the file has been corrupted by a serious error. If you have had a hardware problem such as a power failure or a disk head failure, then the hardware probably caused the corruption. If you have not had any hardware problems, then a software error may have caused the corruption. Note that the /CHECK qualifier does not find all types of file corruption, however.

In either case, you can try using the Convert Utility to fix the problem by using the file specification as both the input-file-spec and the output-file-spec. This operation will reorganize the file; if it does not work, use the Backup Utility (BACKUP) to bring in the backup copy of the file. For more information on both CONVERT and BACKUP, see Sections 10.4.2, 10.5, and 10.6.

Note

If you believe that the software caused the error, submit a Software Performance Report (SPR). Always include the ANALYZE/RMS_FILE check report, a copy of the data file, and a description of what was done with the data file. If possible, also supply a version of the file prior to the corruption and the program or procedure which led to the corruption; being able to reproduce the problem is of tremendous value.

Example 10-1 is a sample Check report of a file with the file specification DISK\$:[HERBER]CUSTDATA.DAT;2.

Example 10-1 Using ANALYZE/RMS_FILE to Create a Check Report

Check RMS File Integrity

14-JUN-1985 21:51:47.38 Page 1

DISK\$: [HERBER]CUSTDATA.DAT;2

FILE HEADER

File Spec: DISK\$: [HERBER]CUSTDATA.DAT;2
File ID: (10044,39,1)
Owner UIC: [011,310]
Protection: System: RWED, Owner: RWED, Group: RWE, World: RWE
Creation Date: 9-JUN-1985 22:30:24.78
Revision Date: 9-JUN-1985 22:30:30.86, Number: 4
Expiration Date: none specified
Backup Date: none posted
Contiguity Options: none
Performance Options: none
Reliability Options: none
Journaling Enabled: none

RMS FILE ATTRIBUTES

File Organization: indexed
Record Format: variable
Record Attributes: carriage-return
Maximum Record Size: 80
Blocks Allocated: 30, Default Extend Size: 2
Bucket Size: 1
Global Buffer Count: 0

FIXED PROLOG

Number of Areas: 8, VBN of First Descriptor: 3
Prolog Version: 3

AREA DESCRIPTOR #0 (VBN 3, offset %X'0000')

Bucket Size: 1
Reclaimed Bucket VBN: 0
Current Extent Start: 1, Blocks: 9, Used: 4, Next: 5
Default Extend Quantity: 2
Total Allocation: 9

AREA DESCRIPTOR #1 (VBN 3, offset %X'0040')

Bucket Size: 1
Reclaimed Bucket VBN: 0
Current Extent Start: 10, Blocks: 3, Used: 1, Next: 11
Default Extend Quantity: 1
Total Allocation: 3

(Continued on next page)

Example 10-1 (Cont.) Using ANALYZE/RMS_FILE to Create a Check Report

AREA DESCRIPTOR #2 (VBN 3, offset %X'0080')

 Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 13, Blocks: 3, Used: 1, Next: 14
 Default Extend Quantity: 1
 Total Allocation: 3

AREA DESCRIPTOR #3 (VBN 3, offset %X'00C0')

 Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 16, Blocks: 3, Used: 1, Next: 17
 Default Extend Quantity: 1
 Total Allocation: 3

AREA DESCRIPTOR #4 (VBN 3, offset %X'0100')

 Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 19, Blocks: 3, Used: 1, Next: 20
 Default Extend Quantity: 1
 Total Allocation: 3

AREA DESCRIPTOR #5 (VBN 3, offset %X'0140')

 Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 22, Blocks: 3, Used: 1, Next: 23
 Default Extend Quantity: 1
 Total Allocation: 3

AREA DESCRIPTOR #6 (VBN 3, offset %X'0180')

 Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 25, Blocks: 3, Used: 1, Next: 26
 Default Extend Quantity: 1
 Total Allocation: 3

AREA DESCRIPTOR #7 (VBN 3, offset %X'01C0')

 Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 28, Blocks: 3, Used: 1, Next: 29
 Default Extend Quantity: 1
 Total Allocation: 3

(Continued on next page)

Example 10-1 (Cont.) Using ANALYZE/RMS_FILE to Create a Check Report

KEY DESCRIPTOR #0 (VBN 1, offset XI'0000')

Next Key Descriptor VBN: 2, Offset: XI'0000'

Index Area: 1, Level 1 Index Area: 1, Data Area: 0

Root Level: 1

Index Bucket Size: 1, Data Bucket Size: 1

Root VBN: 10

Key Flags:

(0)	KEY\$V_DUPKEYS	0
(3)	KEY\$V_IDX_COMPR	0
(4)	KEY\$V_INITIDX	0
(6)	KEY\$V_KEY_COMPR	0
(7)	KEY\$V_REC_COMPR	1

Key Segments: 1

Key Size: 4

Minimum Record Size: 4

Index Fill Quantity: 512, Data Fill Quantity: 512

Segment Positions: 0

Segment Sizes: 4

Data Type: string

Name: "PART_NUM"

First Data Bucket VBN: 4

KEY DESCRIPTOR #1 (VBN 2, offset XI'0000')

Next Key Descriptor VBN: 2, Offset: XI'0066'

Index Area: 3, Level 1 Index Area: 3, Data Area: 2

Root Level: 1

Index Bucket Size: 1, Data Bucket Size: 1

Root VBN: 16

Key Flags:

(0)	KEY\$V_DUPKEYS	1
(1)	KEY\$V_CHGKEYS	0
(2)	KEY\$V_NULKEYS	0
(3)	KEY\$V_IDX_COMPR	0
(4)	KEY\$V_INITIDX	0
(6)	KEY\$V_KEY_COMPR	0

Key Segments: 1

Key Size: 5

Minimum Record Size: 9

Index Fill Quantity: 512, Data Fill Quantity: 512

Segment Positions: 4

Segment Sizes: 5

Data Type: string

Name: "PART_NAME"

First Data Bucket VBN: 13

(Continued on next page)

Example 10-1 (Cont.) Using ANALYZE/RMS_FILE to Create a Check Report

```
KEY DESCRIPTOR #2 (VBN 2, offset %I'0086')
  Next Key Descriptor VBN: 2, Offset: %I'00CC'
  Index Area: 5, Level 1 Index Area: 5, Data Area: 4
  Root Level: 1
  Index Bucket Size: 1, Data Bucket Size: 1
  Root VBN: 22
  Key Flags:
    (0) KEY$V_DUPKEYS      1
    (1) KEY$V_CHGKEYS      0
    (2) KEY$V_NULKEYS      0
    (3) KEY$V_IDX_COMPR    1
    (4) KEY$V_INITIDX      0
    (6) KEY$V_KEY_COMPR    1
  Key Segments: 1
  Key Size: 10
  Minimum Record Size: 19
  Index Fill Quantity: 512, Data Fill Quantity: 512
  Segment Positions:      9
  Segment Sizes:          10
  Data Type: string
  Name: "SUPPLIER_NAME"
  First Data Bucket VBN: 19

KEY DESCRIPTOR #3 (VBN 2, offset %I'00CC')
  Index Area: 7, Level 1 Index Area: 7, Data Area: 6
  Root Level: 1
  Index Bucket Size: 1, Data Bucket Size: 1
  Root VBN: 28
  Key Flags:
    (0) KEY$V_DUPKEYS      1
    (1) KEY$V_CHGKEYS      0
    (2) KEY$V_NULKEYS      0
    (3) KEY$V_IDX_COMPR    1
    (4) KEY$V_INITIDX      0
    (6) KEY$V_KEY_COMPR    1
  Key Segments: 1
  Key Size: 10
  Minimum Record Size: 29
  Index Fill Quantity: 512, Data Fill Quantity: 512
  Segment Positions:      19
  Segment Sizes:          10
  Data Type: string
  Name: "COLOR"
  First Data Bucket VBN: 25
```

The analysis uncovered NO errors.

ANALYZE/RMS_FILE/OUTPUT=CUSTDATA.ANL CUSTDATA.DAT

To place the Check report in a file, use a command of the form

```
ANALYZE/RMS_FILE/CHECK/OUTPUT=output-file-spec input-file-spec
```

The Check report will be placed in the file you named with the output-file-spec parameter. This file will receive the file type ANL by default. For example, the following command will perform an error check on PRLG2.IDX and place the Check report in the file ERROR.ANL.

```
* ANALYZE/RMS_FILE/CHECK/OUTPUT=ERROR PRLG2.IDX
```

10.1.2 Generating a Statistics Report

For indexed files, the Statistics report consists of the Check report plus some additional information about the areas and keys in the file. (A Statistics report on a sequential or relative file is thus the same as a Check report.)

To generate a Statistics report with ANALYZE/RMS_FILE, enter a DCL command of the form

```
ANALYZE/RMS_FILE/STATISTICS file-spec
```

Example 10-2 is an example of a Statistics report.

Example 10-2 Using ANALYZE/RMS_FILE to Create a Statistics Report

RMS File Statistics

18-APR-1985 11:22:27.14 Page 1

DISK\$: [TEST.PROGRAM]INDEX.DAT;1

FILE HEADER

```
File Spec: DISK$: [TEST.PROGRAM]INDEX.DAT;1
File ID: (15960,8,0)
Owner UIC: [011,310]
Protection: System: RWED, Owner: RWED, Group: RWED, World: RWE
Creation Date: 19-APR-1985 22:15:55.70
Revision Date: 19-APR-1985 22:16:01.74, Number: 4
Expiration Date: none specified
Backup Date: 18-APR-1985 00:57:54.24
Contiguity Options: contiguous-best-try
Performance Options: none
Reliability Options: none
Journaling Enabled: none
```

(Continued on next page)

Example 10-2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

RMS FILE ATTRIBUTES

File Organization: indexed
 Record Format: variable
 Record Attributes: carriage-return
 Maximum Record Size: 80
 Blocks Allocated: 30, Default Extend Size: 2
 Bucket Size: 1
 Global Buffer Count: 0

FIXED PROLOG

Number of Areas: 8, VBN of First Descriptor: 3
 Prolog Version: 3

AREA DESCRIPTOR #0 (VBN 3, offset %X'0000')

Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 1, Blocks: 9, Used: 4, Next: 5
 Default Extend Quantity: 2
 Total Allocation: 9

STATISTICS FOR AREA #0

Count of Reclaimed Blocks: 0

AREA DESCRIPTOR #1 (VBN 3, offset %X'0040')

Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 10, Blocks: 3, Used: 1, Next: 11
 Default Extend Quantity: 1
 Total Allocation: 3

STATISTICS FOR AREA #1

Count of Reclaimed Blocks: 0

AREA DESCRIPTOR #2 (VBN 3, offset %X'0080')

Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 13, Blocks: 3, Used: 1, Next: 14
 Default Extend Quantity: 1
 Total Allocation: 3

STATISTICS FOR AREA #2

Count of Reclaimed Blocks: 0

(Continued on next page)

Example 10-2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

AREA DESCRIPTOR #3 (VBN 3, offset 'X'00C0')

Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 16, Blocks: 3, Used: 1, Next: 17
 Default Extend Quantity: 1
 Total Allocation: 3

STATISTICS FOR AREA #3

Count of Reclaimed Blocks: 0

AREA DESCRIPTOR #4 (VBN 3, offset 'X'0100')

Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 19, Blocks: 3, Used: 1, Next: 20
 Default Extend Quantity: 1
 Total Allocation: 3

STATISTICS FOR AREA #4

Count of Reclaimed Blocks: 0

AREA DESCRIPTOR #5 (VBN 3, offset 'X'0140')

Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 22, Blocks: 3, Used: 1, Next: 23
 Default Extend Quantity: 1
 Total Allocation: 3

STATISTICS FOR AREA #5

Count of Reclaimed Blocks: 0

AREA DESCRIPTOR #6 (VBN 3, offset 'X'0180')

Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 25, Blocks: 3, Used: 1, Next: 26
 Default Extend Quantity: 1
 Total Allocation: 3

STATISTICS FOR AREA #6

Count of Reclaimed Blocks: 0

AREA DESCRIPTOR #7 (VBN 3, offset 'X'01C0')

Bucket Size: 1
 Reclaimed Bucket VBN: 0
 Current Extent Start: 28, Blocks: 3, Used: 1, Next: 29
 Default Extend Quantity: 1
 Total Allocation: 3

(Continued on next page)

Example 10-2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

STATISTICS FOR AREA #7

Count of Reclaimed Blocks: 0

KEY DESCRIPTOR #0 (VBN 1, offset %X'0000')

Next Key Descriptor VBN: 2, Offset: %X'0000'

Index Area: 1, Level 1 Index Area: 1, Data Area: 0

Root Level: 1

Index Bucket Size: 1, Data Bucket Size: 1

Root VBN: 10

Key Flags:

(0)	KEY\$V_DUPKEYS	0
(3)	KEY\$V_IDX_COMPR	0
(4)	KEY\$V_INITIDX	0
(6)	KEY\$V_KEY_COMPR	0
(7)	KEY\$V_REC_COMPR	1

Key Segments: 1

Key Size: 4

Minimum Record Size: 4

Index Fill Quantity: 512, Data Fill Quantity: 512

Segment Positions: 0

Segment Sizes: 4

Data Type: string

Name: "ID_NUM"

First Data Bucket VBN: 4

STATISTICS FOR KEY #0

Number of Index Levels:	1
Count of Level 1 Records:	1
Mean Length of Index Entry:	6
Count of Index Blocks:	1
Mean Index Bucket Fill:	4%
Mean Index Entry Compression:	0%
Count of Data Records:	10
Mean Length of Data Record:	33
Count of Data Blocks:	1
Mean Data Bucket Fill:	90%
Mean Data Key Compression:	0%
Mean Data Record Compression:	-2%
Overall Space Efficiency:	2%

(Continued on next page)

Example 10-2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

KEY DESCRIPTOR #1 (VBN 2, offset %X'0000')

Next Key Descriptor VBN: 2, Offset: %X'0006'
 Index Area: 3, Level 1 Index Area: 3, Data Area: 2
 Root Level: 1
 Index Bucket Size: 1, Data Bucket Size: 1
 Root VBN: 16

Key Flags:

(0)	KEY\$V_DUPKEYS	1
(1)	KEY\$V_CHGKEYS	0
(2)	KEY\$V_NULKEYS	0
(3)	KEY\$V_IDX_COMPR	0
(4)	KEY\$V_INITIDX	0
(6)	KEY\$V_KEY_COMPR	0

Key Segments: 1

Key Size: 5

Minimum Record Size: 9

Index Fill Quantity: 512, Data Fill Quantity: 512

Segment Positions: 4

Segment Sizes: 5

Data Type: string

Name: "ID_NAME"

First Data Bucket VBN: 13

STATISTICS FOR KEY #1

Number of Index Levels:	1
Count of Level 1 Records:	1
Mean Length of Index Entry:	7
Count of Index Blocks:	1
Mean Index Bucket Fill:	4%
Mean Index Entry Compression:	0%
Count of Data Records:	6
Mean Duplicates per Data Record:	0
Mean Length of Data Record:	19
Count of Data Blocks:	1
Mean Data Bucket Fill:	24%
Mean Data Key Compression:	0%

(Continued on next page)

Example 10-2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

KEY DESCRIPTOR #2 (VBN 2, offset %X'0066')

Next Key Descriptor VBN: 2, Offset: %X'00CC'

Index Area: 5, Level 1 Index Area: 5, Data Area: 4

Root Level: 1

Index Bucket Size: 1, Data Bucket Size: 1

Root VBN: 22

Key Flags:

(0)	KEY\$V_DUPKEYS	1
(1)	KEY\$V_CHGKEYS	0
(2)	KEY\$V_NULKEYS	0
(3)	KEY\$V_IDX_COMPR	1
(4)	KEY\$V_INITIDX	0
(6)	KEY\$V_KEY_COMPR	1

Key Segments: 1

Key Size: 10

Minimum Record Size: 19

Index Fill Quantity: 512, Data Fill Quantity: 512

Segment Positions: 9

Segment Sizes: 10

Data Type: string

Name: "ADDRESS"

First Data Bucket VBN: 19

STATISTICS FOR KEY #2

Number of Index Levels:	1
Count of Level 1 Records:	1
Mean Length of Index Entry:	12
Count of Index Blocks:	1
Mean Index Bucket Fill:	4%
Mean Index Entry Compression:	58%
Count of Data Records:	7
Mean Duplicates per Data Record:	0
Mean Length of Data Record:	20
Count of Data Blocks:	1
Mean Data Bucket Fill:	30%
Mean Data Key Compression:	21%

(Continued on next page)

Example 10-2 (Cont.) Using ANALYZE/RMS_FILE to Create a Statistics Report

KEY DESCRIPTOR #3 (VBN 2, offset %X'OCCC')

Index Area: 7, Level 1 Index Area: 7, Data Area: 6
 Root Level: 1
 Index Bucket Size: 1, Data Bucket Size: 1
 Root VBN: 28
 Key Flags:

(0)	KEY\$V_DUPKEYS	1
(1)	KEY\$V_CHGKEYS	0
(2)	KEY\$V_NULKEYS	0
(3)	KEY\$V_IDX_COMPR	1
(4)	KEY\$V_INITIDX	0
(6)	KEY\$V_KEY_COMPR	1

Key Segments: 1
 Key Size: 10
 Minimum Record Size: 20
 Index Fill Quantity: 512, Data Fill Quantity: 512
 Segment Positions: 19
 Segment Sizes: 10
 Data Type: string
 Name: "CHARGES"
 First Data Bucket VBN: 25

STATISTICS FOR KEY #3

Number of Index Levels:	1
Count of Level 1 Records:	1
Mean Length of Index Entry:	12
Count of Index Blocks:	1
Mean Index Bucket Fill:	4%
Mean Index Entry Compression:	58%
Count of Data Records:	5
Mean Duplicates per Data Record:	1
Mean Length of Data Record:	23
Count of Data Blocks:	1
Mean Data Bucket Fill:	25%
Mean Data Key Compression:	34%

The analysis uncovered NO errors.

ANALYZE/RMS_FILE/OUTPUT=INDEX/STATISTICS INDEX.DAT

10.1.3 Using Interactive Mode

The `/INTERACTIVE` qualifier begins an interactive session in which you can examine the structure of a VAX RMS file.

`ANALYZE/RMS_FILE` imposes a hierarchical tree structure on the internal VAX RMS file structure. Each data structure in the file is a node, with a branch for each pointer in the data structure. The file header is always the root node. Each of the three file organizations (sequential, relative, and indexed) has its own tree structure.

To examine a file, you enter commands that move the current position to particular structures within the tree. The utility displays the current structure on the screen.

Table 10-1 summarizes the `ANALYZE/RMS_FILE` commands.

Table 10-1 ANALYZE/RMS_FILE Command Summary

Command	Function
AGAIN	Displays the current structure again.
DOWN [branch]	Moves the structure pointer down to the next level. If the current node has more than one branch, the branch keyword must be specified. If a branch keyword is required but not specified, the utility will display a list of possibilities to prompt you. You can also display the list by specifying "DOWN ?."
DUMP n	Displays a hexadecimal dump of the specified block.
EXIT	Ends the interactive session.
FIRST	Moves the structure pointer to the first structure on the current level. The structure is displayed. For example, if you are examining data buckets and want to examine the first bucket, this command will put you there and display the first bucket's header.
HELP [keyword ...]	Displays help messages about the interactive commands.
NEXT	Moves the structure pointer to the next structure on the current level. The structure is displayed. Pressing the RETURN key is equivalent to a NEXT command.

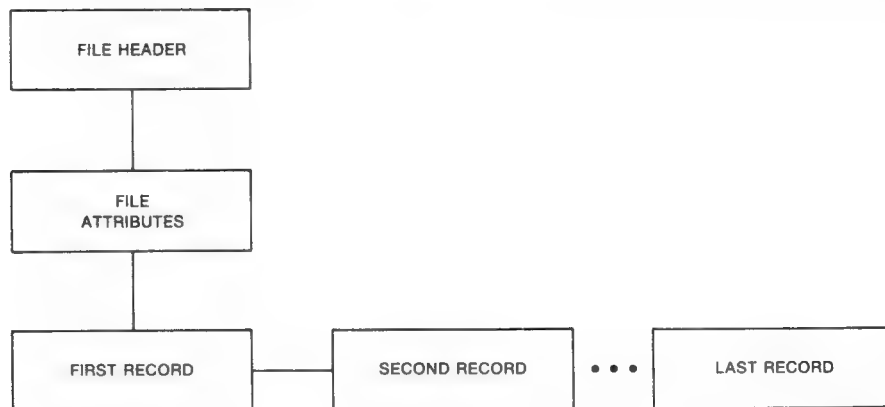
Table 10-1 (Cont.) ANALYZE/RMS_FILE Command Summary

Command	Function
REST	Moves the structure pointer along the rest of the structures on the current level, and each is displayed in turn.
TOP	Moves the structure pointer up to the file header. The file header is displayed.
UP	Moves the structure pointer up to the next level. The structure at that level is displayed.

10.1.4 Examining a Sequential File

Figure 10-1 shows the tree structure of a sequential file.

Figure 10-1 Tree Structure for Sequential Files



ZK-327-81

The FILE HEADER structure is always the first structure displayed. From the FILE HEADER structure, a DOWN command moves the current position to the FILE ATTRIBUTES structure. A DOWN command from the FILE ATTRIBUTES structure moves the current position to the first record in the file. From the first record, the REST command will move the current position through the records in the file, displaying each one in turn. A series of NEXT commands will also accomplish this same operation.

Figure 10-2 shows the layout and contents of the records in a sequential file SEQ.DAT. Example 10-3 is an interactive examination of SEQ.DAT, showing the contents of three records in the file.

Figure 10-2 Record Layout and Content for SEQ.DAT

[illegible]

Example 10-3 Examining a Sequential File

\$ ANALYZE/RMS_FILE/INTERACTIVE SEQ.DAT

FILE HEADER

File Spec: DISK\$DELPHIWORK:[RMS32]SEQ.DAT;3
 File ID: (1170,2,2)
 Owner UIC: [730,465]
 Protection: System: RWED, Owner: RWED, Group: RWED, World:
 Creation Date: 7-MAY-1985 16:51:30.92
 Revision Date: 8-MAY-1985 14:02:17.15, Number: 3
 Expiration Date: none specified
 Backup Date: none posted
 Contiguity Options: none
 Performance Options: none
 Reliability Options: none

ANALYZE> DOWN

RMS FILE ATTRIBUTES

File Organization: sequential
 Record Format: variable
 Record Attributes: carriage-return
 Maximum Record Size: 0
 Longest Record: 73
 Blocks Allocated: 3, Default Extend Size: 0
 End-of-File VBN: 1, Offset: %X'00E4'

ANALYZE> DOWN

DATA BYTES (VBN 1, offset %X'0000'):															
	7	6	5	4	3	2	1	0							
										01234567					
31	30	30	30	30	30	00	49		0000	I.000001					
20	4C	41	54	49	47	49	44		0008	DIGITAL					
4E	45	4D	50	49	55	51	45		0010	EQUIPMEN					
52	4F	50	52	4F	43	20	54		0018	T CORPOR					
31	31	20	4E	4F	49	54	41		0020	ATION 11					
42	20	54	49	50	53	20	30		0028	O SPIT B					
41	4F	52	20	4B	4F	4F	52		0030	ROOK ROA					
41	55	48	53	41	4E	20	44		0038	D NASHUA					
33	30	48	4E	20	20	20	20		0040	NH03					
									0048	061.					

(Continued on next page)

Example 10-3 (Cont.) Examining a Sequential File

ANALYZE> NEXT

DATA BYTES (VBN 1, offset XI'004C'):

7	6	5	4	3	2	1	0		01234567
32	30	30	30	30	30	00	49	0000	I.000002
49	46	46	4F	20	42	44	41	0008	ADB OFFI
4C	50	50	55	53	20	45	43	0010	CE SUPPL
20	20	20	20	20	53	45	49	0018	IES
32	34	20	20	20	20	20	20	0020	42
4F	4D	45	53	4F	52	20	30	0028	O ROSEMO
45	52	54	53	20	54	4E	55	0030	UNT STRE
49	44	20	4E	41	53	54	45	0038	ETSAN DI
32	39	41	43	20	4F	47	45	0040	EGO CA92
				00	30	31	31	0048	110.

ANALYZE> NEXT

DATA BYTES (VBN 1, offset XI'0098'):

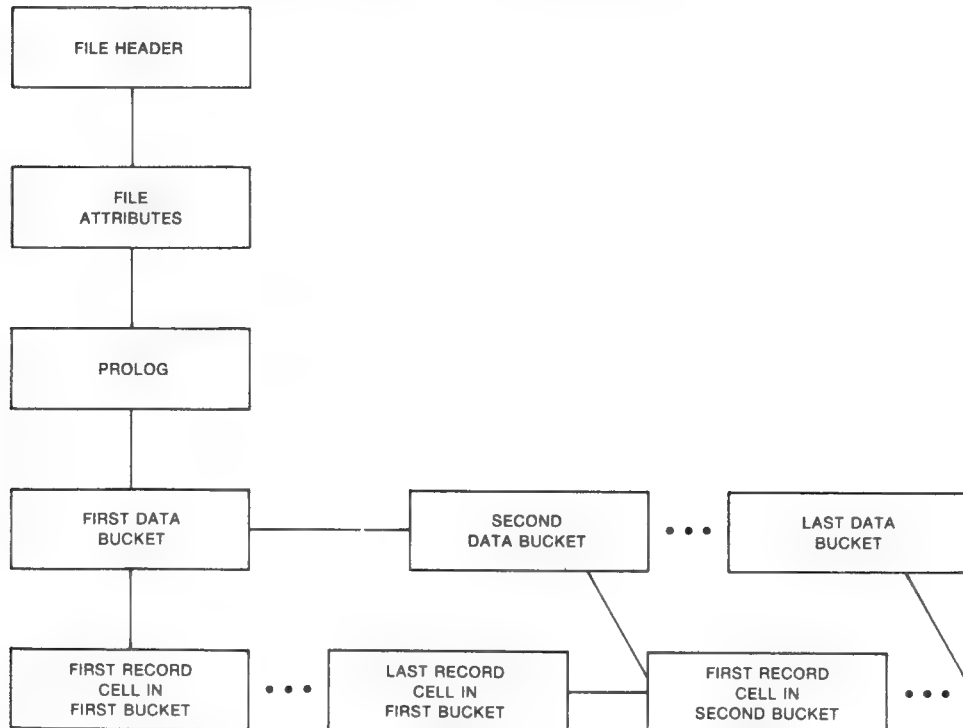
7	6	5	4	3	2	1	0		01234567
33	30	30	30	30	30	00	49	0000	I.000003
52	50	20	52	4F	4C	4F	43	0008	COLOR PR
4C	20	47	4E	49	54	4E	49	0010	INTING L
52	4F	54	41	52	4F	42	41	0018	LABORATOR
34	39	20	20	20	53	45	49	0020	IES 94
35	20	54	53	41	45	20	39	0028	9 EAST 5
45	45	52	54	53	20	48	54	0030	TH STREE
4F	59	20	57	45	4E	20	54	0038	T NEW YO
30	31	59	4E	20	20	4B	52	0040	RK NY10
				00	33	30	30	0048	003.

ANALYZE> EXIT

10.1.5 Examining a Relative File

Figure 10-3 shows the tree structure for relative files.

Figure 10-3 Tree Structure of Relative Files



ZK-328-81

The tree structure for relative files also begins with the FILE HEADER and FILE ATTRIBUTES structures. From the FILE ATTRIBUTES structure, the next structure down is the PROLOG. The first structure down from the PROLOG is the first DATA BUCKET. The DATA BUCKET structures can be examined with the REST command or one-by-one with the NEXT command. The only information at the DATA BUCKET level is the number of the data bucket's virtual block.

From the first DATA BUCKET structure, the next structure down is the first CELL AND RECORD IN FIRST BUCKET. You can examine the records in each cell by specifying either the REST command or a series of NEXT commands.

Example 10-4 shows an interactive examination of a relative file.

Example 10-4 Examining a Relative File

FILE HEADER

File Spec: DISK\$NEWWORK:[RMS32]REL.DAT;1
File ID: (9573,7,2)
Owner UIC: [181,065]
Protection: System: RWED, Owner: RWED, Group: RE, World:
Creation Date: 22-MAY-1982 10:42:04.95
Revision Date: 22-MAY-1982 10:42:05.81, Number: 1
Expiration Date: none specified
Backup Date: none posted
Contiguity Options: contiguous-best-try
Performance Options: none
Reliability Options: none

ANALYZE> DOWN

RMS FILE ATTRIBUTES

File Organization: relative
Record Format: variable
Record Attributes: carriage-return
Maximum Record Size: 75
Blocks Allocated: 9, Default Extend Size: 0
Bucket Size: 3
Global Buffer Count: 0

ANALYZE> DOWN

FIXED PROLOG

Prolog Flags:
(0) PLG\$V_NOEXTEND 0
First Data Bucket VBN: 2
Maximum Record Number: 2147483647
End-of-File VBN: 10
Prolog Version: 1

ANALYZE> DOWN

DATA BUCKET (VBN 2)

(Continued on next page)

Example 10-4 (Cont.) Examining a Relative File

ANALYZE> DOWN

RECORD CELL (VBN 2, offset %X'0000'):

Cell Control Flags:

(2) DLC\$V_DELETED 0

(3) DCL\$V_REC 1

Record Bytes:

7	6	5	4	3	2	1	0		01234567
31	30	30	30	30	30	00	49	0000	I.000001
20	4C	41	54	49	47	49	44	0008	DIGITAL
4E	45	4D	50	49	55	51	45	0010	EQUIPMEN
52	4F	50	52	4F	43	20	54	0018	T CORPOR
31	31	20	4E	4F	49	54	41	0020	ATION 11
42	20	54	49	50	53	20	30	0028	O SPIT B
41	4F	52	20	4B	4F	4F	52	0030	ROOK ROA
41	55	48	53	41	4E	20	44	0038	D NASHUA
33	30	48	4E	20	20	20	20	0040	NH03
				31	36	30		0048	061

If you use the REST command at the CELL AND RECORD level, the utility will display all the cells and records in the file, not just the cells and records in the current bucket.

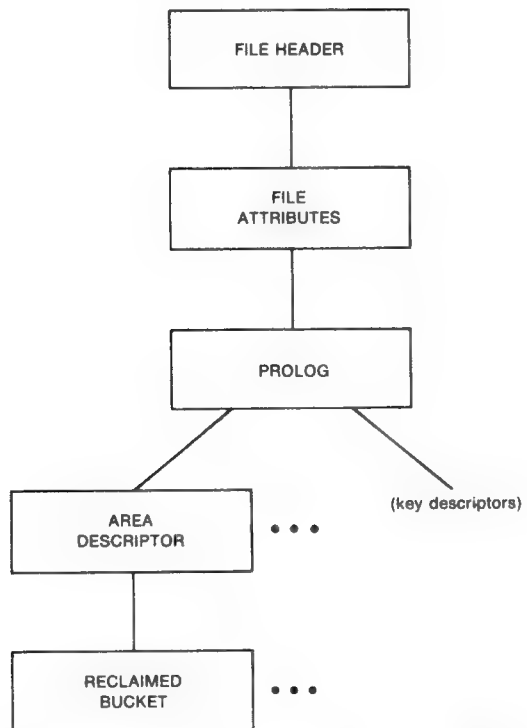
10.1.6 Examining an Indexed File

The structure of an indexed file also begins with the FILE HEADER, FILE ATTRIBUTES, and PROLOG structures. From the PROLOG structure, the file structure branches to the AREA DESCRIPTORS and the KEY DESCRIPTORS. To branch to the AREA DESCRIPTOR path, specify the command DOWN AREA. To branch to the KEY DESCRIPTOR path, specify DOWN KEY.

The AREA DESCRIPTOR path contains structures that show information about the various areas in the file. The KEY DESCRIPTOR path contains the primary key structures (and data records) and any secondary key structures.

Figure 10-4 shows the structure following the AREA DESCRIPTOR path.

Figure 10-4 AREA DESCRIPTOR Path



ZK-329-81

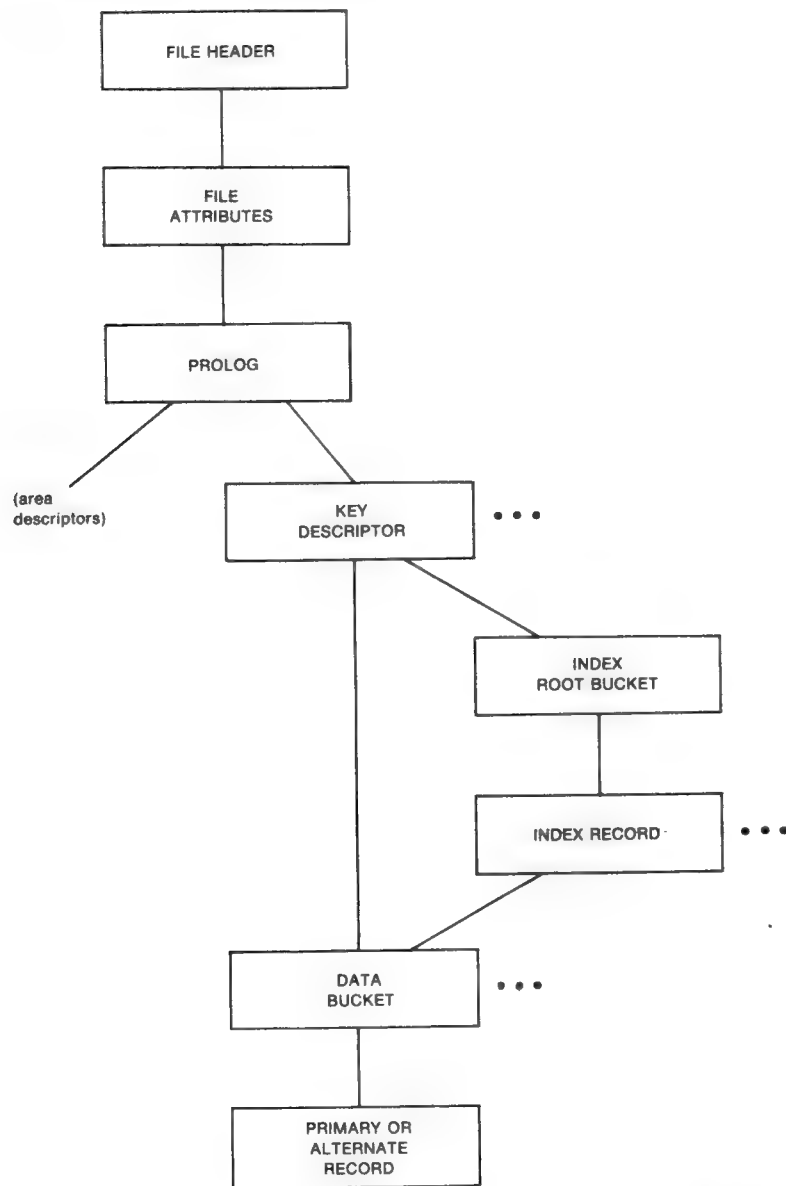
Example 10-5 is an example of an examination of an AREA DESCRIPTOR from the PROLOG level.

Example 10-5 Examining an AREA DESCRIPTOR

```
ANALYZE> DOWN AREA
AREA DESCRIPTOR #0 (VBN 3, offset %X'0000')
  Bucket Size: 1
  Alignment: AREA$C_NONE
  Alignment Flags:
    (0) AREA$V_HARD      0
    (1) AREA$V_ONC       0
    (5) AREA$V_CBT       0
    (7) AREA$V_CTG       0
  Current Extent Start: 1, Blocks: 9, Used: 7, Next: 8
  Default Extend Quantity: 0
```

Figure 10-5 shows the structure following the KEY DESCRIPTOR path.

Figure 10-5 KEY DESCRIPTOR Path



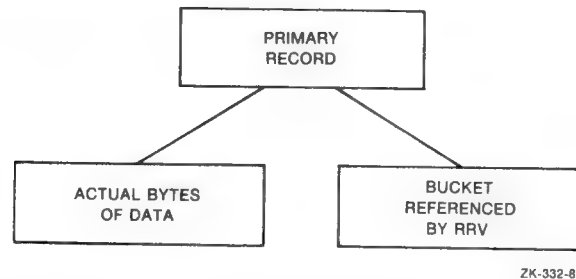
ZK-330-81

As shown in Figure 10-5, you can branch directly to the DATA BUCKET, or you can branch to the INDEX BUCKET to begin examination of the index structure, eventually reaching the DATA BUCKET structure. Depending on whether you are examining the primary index structure or one of the alternate index structures, there is a difference in the contents of the RECORD structure.

The primary record structure contains the actual data records; the alternate record structures contain secondary index data records (SIDRs).

Figure 10-6 displays the structure of the primary records.

Figure 10-6 Structure of Primary Records

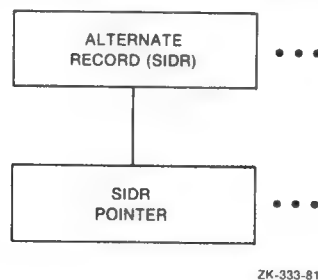


As shown in Figure 10-6, the branch from the primary record structure allows you to either examine the actual bytes of data within the record or to follow the RRV.

Example 10-6 shows an examination of a primary record.

Figure 10-7 displays the structure of the alternate records.

Figure 10-7 Structure of Alternate Records



Example 10-6 Examining a Primary Record

PRIMARY DATA RECORD (VBN 4, offset %X'000E')

Record Control Flags:

```
(2) IRC$V_DELETED    0
(3) IRC$V_RRV        0
(4) IRC$V_NOPTBSZ    0
```

Record ID: 1

RRV ID: 1, 4-Byte Bucket Pointer: 4

Key:

```
7 6 5 4 3 2 1 0      01234567
-----
31 30 30 30 30 30 0000 | 0000001 |
```

ANALYZE> DOWN BYTES

```
7 6 5 4 3 2 1 0      01234567
-----
31 30 30 30 30 30 00 49 | 0000 | I.000001 |
20 4C 41 54 49 47 49 44 | 0008 | DIGITAL |
4E 45 4D 50 49 55 51 45 | 0010 | EQUIPMEN |
52 4F 50 52 4F 43 20 54 | 0018 | T CORPOR |
31 31 20 4E 4F 49 54 41 | 0020 | ATION 11 |
42 20 54 49 50 53 20 30 | 0028 | O SPIT B |
41 4F 52 20 4B 4F 4F 52 | 0030 | BOOK ROA |
41 55 48 53 41 4E 20 44 | 0038 | D NASHUA |
33 30 48 4E 20 20 20 20 | 0040 | NHOS |
31 36 30 | 0048 | 061 |
```

ANALYZE> UP

PRIMARY DATA RECORD (VBN 4, offset %X'000E')

Record Control Flags:

```
(2) IRC$V_DELETED    0
(3) IRC$V_RRV        0
(4) IRC$V_NOPTBSZ    0
```

Record ID: 1

RRV ID: 1, 4-Byte Bucket Pointer: 4

Key:

```
7 6 5 4 3 2 1 0      01234567
-----
31 30 30 30 30 30 0000 | 0000001 |
```

ANALYZE> DOWN RRV

BUCKET HEADER (VBN 4)

Check Character: %X'00'

Area Number: 0

VBN Sample: 4

Free Space Offset: %X'0104'

Free Record ID Range: 4 - 255

Next Bucket VBN: 4

Level: 0

Bucket Header Flags:

```
(0) BKT$V_LASTBKT    1
(1) BKT$V_ROOTBKT    0
```

Example 10-7 shows an examination of an alternate record.

Example 10-7 Examining an Alternate Record

```

ANALYZE> DOWN
SIDR RECORD (VBN 6, offset %X'000E')
  Control Flags:
    (4) IRC$V_NOPTRSZ    0
  Record ID: 1
  Key:
      7 6 5 4 3 2 1 0      01234567
      -----
      31 36 30 33 30| 0000 |03061 |
ANALYZE> DOWN
  sidr pointer control flags:
    (2) IRC$V_DELETED    0
    (6) IRC$V_KEYDELETE  0
  sidr pointer record id: 1, 4-byte record VBN: 4

```

10.2 Generating an FDL File from a Data File

You can use the Analyze/RMS_File Utility to create an FDL file generally called an *analysis file*. FDL files created by ANALYZE/RMS_FILE contain statistics about each area and key in the primary sections named ANALYSIS_OF_AREA and ANALYSIS_OF_KEY.

These analysis sections are then used by the Edit/FDL Utility in its Optimize script. You can compare the statistics in these sections with your assumptions about the file's use; you may find some places in the file's structure where additional tuning will be possible.

To generate an FDL file from a data file, give this command:

```
ANALYZE/RMS_FILE/FDL file-spec
```

With a command of this type, the FDL file would take its file-name from the input file-spec; to assign a different file-name, use the /OUTPUT qualifier. For example, the following command would generate an FDL file named INDEXDEF.FDL from the data file CUSTFILE.DAT:

```
$ ANALYZE/RMS_FILE/FDL/OUTPUT=INDEXDEF CUSTFILE.DAT
```

Example 10-8 illustrates an FDL file showing the KEY and ANALYSIS_OF_KEY sections for an indexed file with two keys.

Example 10-8 KEY and ANALYSIS_OF_KEY Sections in an FDL File

```

IDENT      2-JUN-1985 16:15:35      VAX/VMS ANALYZE/RMS_FILE Utility
SYSTEM
FILE
SOURCE          VAX/VMS
ALLOCATION        9
BEST_TRY_CONTIGUOUS  no
BUCKET_SIZE      1
CONTIGUOUS       no
EXTENSION        0
GLOBAL_BUFFER_COUNT 0
NAME             DISK$USERWORK: [WORK.RMS32] CUSTDATA.DAT;4
ORGANIZATION     indexed
OWNER            [520,50]
PROTECTION       (system:RWED, owner:RWED, group:RWED, world:)
READ_CHECK       no
WRITE_CHECK      no
RECORD
BLOCK_SPAN       yes
CARRIAGE_CONTROL carriage_return
FORMAT           variable
SIZE             0
AREA 0
ALLOCATION        9
BEST_TRY_CONTIGUOUS  no
BUCKET_SIZE      1
CONTIGUOUS       no
EXTENSION        0
KEY 0
CHANGES         no
DATA_AREA        0
DATA_FILL        100
DUPLICATES       no
INDEX_AREA       0
INDEX_FILL       100
LEVEL1_INDEX_AREA 0
NULL_KEY         no
PROLOG           1
SEGO_LENGTH      0
SEGO_POSITION    0
TYPE             string

```

(Continued on next page)

Example 10-8 (Cont.) KEY and ANALYSIS_OF_KEY Sections in an FDL File

```

KEY 1
  CHANGES          no
  DATA_AREA        0
  DATA_FILL        100
  DUPLICATES        yes
  INDEX_AREA        0
  INDEX_FILL        100
  LEVEL1_INDEX_AREA 0
  NULL_KEY          no
  SEGO_LENGTH        5
  SEGO_POSITION      68
  TYPE              string

ANALYSIS_OF_AREA 0
  RECLAIMED_SPACE    0

ANALYSIS_OF_KEY 0
  DATA_FILL          50
  DATA_RECORD_COUNT  3
  DATA_SPACE_OCCUPIED 1
  DEPTH               1
  INDEX_FILL          4
  INDEX_SPACE_OCCUPIED 1
  MEAN_DATA_LENGTH    73
  MEAN_INDEX_LENGTH   9

ANALYSIS_OF_KEY 1
  DATA_FILL          14
  DATA_RECORD_COUNT  3
  DATA_SPACE_OCCUPIED 1
  DEPTH               1
  DUPLICATES_PER_SIDR 1
  INDEX_FILL          4
  INDEX_SPACE_OCCUPIED 1
  MEAN_DATA_LENGTH    19
  MEAN_INDEX_LENGTH   8
  
```

10.3 Optimizing and Redesigning File Characteristics

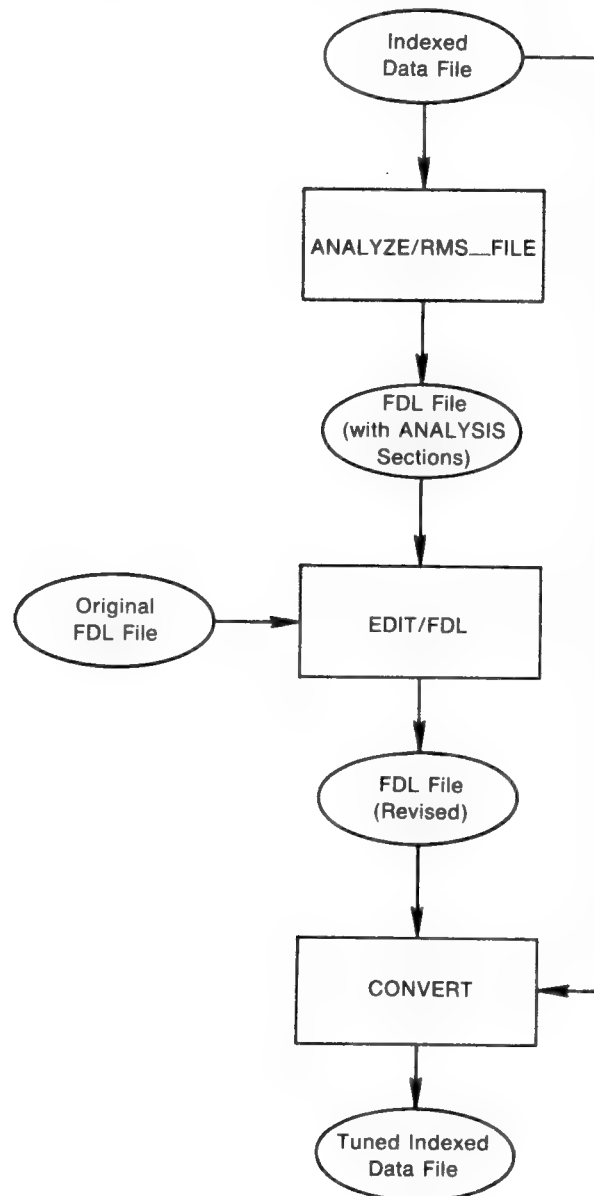
To maintain your files properly, you must occasionally tune them. Tuning involves adjusting and readjusting the characteristics of the file, generally to make the file run faster or more efficiently, and then reorganizing the file to reflect those changes.

There are basically two ways to tune files. You can redesign your FDL file to change file characteristics or parameters. You can change these characteristics either interactively with EDIT/FDL (the preferred method) or by using a text editor. With the redesigned FDL file, then, you can create a new data file.

You can also optimize your data file by using ANALYZE/RMS_FILE with the /FDL qualifier. This method, rather than actually redesigning your FDL file, produces an FDL file containing certain statistics about the file's use that you can then use to tune your existing data file.

Figure 10-8 shows how to use the VAX RMS utilities to perform the tuning cycle.

Figure 10-8 The VAX RMS Tuning Cycle



ZK-952-82

Section 10.3.1 describes how to redesign an FDL file, and Section 10.3.2 explains how to optimize the run-time performance of a data file.

10.3.1 Redesigning an FDL File

There are many ways to redesign an FDL file. If you want to make small changes, you can use the ADD, DELETE, and MODIFY commands at the main menu (main editor) level.

Command	Function
ADD	<p>Allows you to add one or more new lines to the FDL file. When you give the ADD command at the main menu level, EDIT/FDL prompts you with a menu displaying all legal primary attributes; your FDL file does not necessarily have to contain all these attributes. You can add a new primary attribute to your file, or you can add a new secondary attribute to an existing primary attribute.</p> <p>When you type in a primary attribute, EDIT/FDL displays all the legal secondary attributes for that primary attribute with their possible values. You can then select the secondary attribute that you want to add to your FDL file and supply the appropriate value for the secondary attribute.</p>
DELETE	<p>Allows you to delete one or more lines from the FDL file. When you give the DELETE command at the main menu level, EDIT/FDL prompts you with a menu displaying the current primary attributes of your FDL file.</p> <p>When you select the primary attribute that has the secondary attribute you want to remove from your current FDL definition, EDIT/FDL displays all the existing secondary attributes of your FDL file with their current values. When you select the secondary attribute, EDIT/FDL removes it from the FDL definition. Also, when you delete the last secondary attribute of a particular primary attribute, EDIT/FDL also removes the primary attribute from the current definition.</p>
MODIFY	<p>Allows you to change an existing line in the FDL definition. When you issue the MODIFY command at the main menu level, EDIT/FDL prompts you with a menu displaying the current primary attributes of your FDL file.</p> <p>When you type in a primary attribute, EDIT/FDL displays all the existing secondary attributes for that primary attribute with their current values. You can then select the secondary attribute of which you want to change the value and supply the appropriate value for the secondary attribute.</p>

However, if you want to make substantial changes to an FDL file, you should invoke the Touch-up script. Because sequential and relative files are simple in design, the Touch-up script works only with FDL files that describe indexed files. If you want to redesign sequential and relative files, you can use the command listed above (ADD, DELETE, or MODIFY), or you can go through the design phase again, using the scripts for those organizations.

To completely redesign an existing FDL file that describes an indexed sequential file, specify a DCL command of the form

```
EDIT/FDL/SCRIPT=TOUCHUP fdl-file-spec
```

The script that follows is similar to the Indexed script.

10.3.2 Optimizing a Data File

You can optimize the run-time performance of an existing data file either interactively or noninteractively. In either case, however, the first step is to create an FDL file from the data file by using the Analyze/RMS_File Utility. This analysis produces an FDL file that includes the ANALYSIS_OF_AREA and ANALYSIS_OF_KEY sections.

If you then want to optimize your data file interactively by going through the Optimize script, use a command of the form

```
EDIT/FDL/ANALYZE=fdl-file-spec/SCRIPT=OPTIMIZE new-fdl-file-spec
```

The FDL file-spec that appears with the /ANALYZE qualifier is the file specification of the FDL file created with ANALYZE/RMS_FILE. The new-fdl-file-spec is the name of the new, optimal FDL file that you are creating.

The script from that point is similar to an Indexed, Relative, or Sequential script.

If you want to optimize an existing FDL file but do not wish to go through an interactive session, you can use a command of the form

```
EDIT/FDL/ANALYZE=fdl-file-spec/NOINTERACTIVE new-fdl-file-spec
```

The final stage of optimization is to use the Convert Utility with the old data file and the new FDL file to create a new, optimal data file:

```
CONVERT/FDL=new-fdl-file old-file new-file
```

10.4 Making a File Contiguous

If your file has been in use for some time or if it is extremely volatile, the numerous deletions and insertions of records may have caused the optimal design of the file to deteriorate. For example, numerous extensions will degrade performance by causing window-turn operations. In indexed files, deletions can cause empty but unusable buckets to accumulate.

If additions or insertions to a file cause too many extensions, the file's performance will also deteriorate. To improve performance, you could increase the file's window size, but this uses an expensive system resource and at some point may itself hurt performance. A better method is to make the file contiguous again.

This section presents techniques for cleaning up your files. These techniques include using the Copy, Convert, and Convert/Reclaim utilities.

10.4.1 Using the Copy Utility

You can use the COPY command with the /CONTIGUOUS qualifier to copy the file, creating a new contiguous version. The /CONTIGUOUS qualifier can be used only on an output file.

To use the COPY command with the /CONTIGUOUS qualifier, give a command of the form

```
COPY input-file-spec output-file-spec/CONTIGUOUS
```

If you do not want to rename the file, use the same file-spec for both input and output.

By default, if the input file is contiguous, COPY likewise tries to create a contiguous output file. By using the /CONTIGUOUS qualifier, you ensure that the output file is copied to consecutive physical disk blocks.

The /CONTIGUOUS qualifier can only be used when you copy disk files; it does not apply to tape files. For more information, see the COPY command in the *VAX/VMS DCL Dictionary*.

10.4.2 Using the Convert Utility

The Convert Utility can also make a file contiguous if contiguity were an original attribute of the file.

To use the Convert Utility to make a file contiguous, give a command of the form

```
CONVERT input-file-spec output-file-spec
```

Again, if you do not want to rename the file, use the same file-spec for both input and output.

10.4.3 Reclaiming Buckets in Prolog 3 Files

If you have deleted a number of records from a Prolog 3 indexed file, it is possible that you have deleted all of the data entries in a particular bucket. VAX RMS generally cannot use such empty buckets to write new records.

With Prolog 3 indexed files, you can reclaim such buckets by using the Convert/Reclaim Utility. This utility allows you to reclaim the buckets without incurring the overhead of reorganizing the file with CONVERT.

As the data buckets are reclaimed, the pointers to them in the index buckets are deleted. If as a result any of the index buckets become empty, they too are reclaimed.

Note that RFA access is retained after bucket reclamation. The only effect that CONVERT/RECLAIM has on a Prolog 3 indexed file is that empty buckets are reclaimed.

To use CONVERT/RECLAIM, give a command of the following form, where file-spec specifies a Prolog 3 indexed file:

```
CONVERT/RECLAIM file-spec
```

Please note that the file cannot be open for shared access at the time that you give the CONVERT/RECLAIM command.

10.5 Reorganizing a File

Using the Convert Utility is the easiest way to reorganize a file. In addition, CONVERT cleans up split buckets in indexed files. Also, because the file is completely reorganized, buckets in which all the records were deleted will disappear. (Note that this is not the same as bucket reclamation. With CONVERT, the file becomes a new file and records receive new RFAs.)

To use the Convert Utility to reorganize a file, give a command of this form:

```
CONVERT input-file-spec output-file-spec
```

Again, if you do not want to rename the file, use the same file-spec for both input and output.

10.6 Making Archive Copies

Another part of maintaining files is making sure that the valuable data in them is protected. You should keep duplicates of your files in some other place in case something happens to the originals. In other words, you need to *back up* your files. Then, if something does happen to your original data, you can restore the duplicate files.

The Backup Utility (BACKUP) allows you to create backup copies of files and directories and to restore them, as well. These backup copies are called *save sets*, and they can reside on either disk or magnetic tape. Save sets are also written in BACKUP format; only BACKUP can interpret the data.

Unlike the DCL command COPY, which makes new copies of file (updating the revision dates and assigning protection from the defaults that apply), BACKUP makes copies that are identical in all respects to the originals, including dates and protection.

To use the Backup Utility to create a save set of your file, give a command of the form

```
BACKUP input-file-spec output-file-spec[/SAVE_SET]
```

You have to use the /SAVE_SET qualifier only if the output file will be backed up to disk. You can omit the qualifier for magnetic tape.

For more information on BACKUP, see the description of the Backup Utility in the *VAX/VMS Backup Utility Reference Manual*.

A

EDIT / FDL Optimization Algorithms

This appendix lists the algorithms used by the Edit/FDL Utility to determine the optimum values for file attributes.

A.1 Allocation

For sequential files with block spanning, EDIT/FDL allocates enough blocks to hold the specified number of records of mean size. If you do not allow block spanning, EDIT/FDL factors in the potential wasted space at the end of each block.

For relative files, EDIT/FDL calculates the total number of buckets in the file and then allocates enough blocks to hold the required number of buckets.

For indexed files, EDIT/FDL calculates the depth, in order to determine the actual bucket size and number of buckets at each level of the index. It then allocates enough blocks to hold the required number of buckets. Areas for the data level (Level 0) have separate allocations from the areas for the index levels of each key.

In all cases, allocations are rounded up. This algorithm does not allocate as many "extra" blocks as it did in VAX/VMS Version 3.0. Allocation is now a multiple of bucket size.

A.2 Extension Size

For all file organizations, EDIT/FDL sets the extension size to one-tenth of the allocation size and truncates any fraction. This algorithm does not allocate as many "extra" blocks as it did in VAX/VMS Version 3.0.

A.3 Bucket Size

For relative files, the bucket size calculation depends on the intended use of the file. If you have instructed EDIT/FDL that successive accesses to the file will be clustered among records that are close to each other, the bucket size will be large enough for 16 records. In all other cases, bucket size will be large enough for 4 records. The maximum bucket size is 63 blocks.

For indexed files, EDIT/FDL permits you to decide the bucket size for any particular index. The data and index levels get the same bucket size but you can use the MODIFY command to change these values.

EDIT/FDL calculates the default bucket size by first finding the most common index depth produced by the various bucket sizes. If you specify smaller buffers rather than fewer levels, EDIT/FDL establishes the default bucket size as the smallest size needed to produce the most common depth. On Surface_Plot graphs, these values are shown on the leftmost edge of each bucket size.

Note

If you specify a separate bucket size for the Level 1 index level, it should match the bucket size assigned to the rest of the index.

The bucket size in VAX/VMS Version 4 is always a multiple of disk cluster size. The ANALYZE/RMS_FILE primary attribute ANALYSIS_OF_KEY now has a new secondary attribute called LEVEL1_RECORD_COUNT that represents the index level immediately above the data. It makes the tuning algorithm more accurate when duplicate key values are specified.

A.4 Global Buffers

The global buffer count is the number of I/O buffers that two or more processes can access. In VAX/VMS Version 3, this algorithm allocated global buffers equal to two times the number of processes trying to share the file. Now, however, it tries to cache or "map" the whole Key 0 index (at least up to a point) into memory for quicker and more efficient access.

A.5 Index Depth

The indexed design routines simulate the loading of data buckets with records based on your data regarding key sizes, key positions, record sizes (mean and maximum), compression values, load method, and fill factors.

When EDIT/FDL finds the number of required data buckets, it can determine the actual number of index records in the next level up (each of which points to a data bucket). The process is repeated until all the required index records for a level can fit in one bucket. (This is the root.) If your file exceeds 32 levels, EDIT/FDL issues an error message.

With a `line_plot`, the design calculations are performed up to 63 times—once for each legal bucket size. With a `surface_plot`, each line of the plot is equivalent to a `line_plot` with a different value for the variable on the Y-axis.

Glossary

alternate key: An optional key within the data records in an indexed file; used by VAX RMS to build an alternate index. See *key (indexed files)* and *primary key*.

area: Areas are VAX RMS-maintained regions of an indexed file. They allow a user to specify placement and/or specific bucket sizes for particular portions of a file. An area consists of any number of buckets, and there may be from 1 to 255 areas in a file.

asynchronous record operation: An operation in which your program may possibly regain control before the completion of a record retrieval or storage request. Completion AST's and the Wait service are the mechanisms provided by VAX RMS for programs to synchronize with asynchronous record operations. See *synchronous record operation*.

bits per inch: The recording density of a magnetic tape. Indicates how many characters can fit on one inch of the recording surface. See *density*.

block: A group of consecutive bytes of data treated as a unit by the storage medium. A block on a Files-11 On-Disk Structure disk is 512 eight-bit bytes.

block I/O: The set of VAX RMS procedures that allow you direct access to the blocks of a file regardless of file organization.

bootstrap block: A block in the index file of a system disk. Can contain a program that loads the operating system into memory.

bpi: See *bits per inch*.

bucket: A storage structure, consisting of from 1 to 32 blocks, used for building and processing relative and indexed files. A bucket contains one or more records or record cells. Buckets are the unit of contiguous transfer between VAX RMS buffers and the disk.

bucket split: The result of inserting records into a full bucket. To minimize bucket splits, VAX RMS attempts to keep half of the records in the original bucket and transfers the remaining records to a newly created bucket.

buffer: An internal memory area used for temporary storage of data records during input or output operations.

cluster: The basic unit of space allocation on a Files-11 On-Disk Structure volume. Consists of one or more contiguous blocks, with the number being specified when the volume is initialized.

contiguous area: A group of physically adjacent blocks.

cylinder: The tracks at the same radius on all recording surfaces of a disk.

density: The number of bits per inch of magnetic tape. Typical values are 800 bpi and 1600 bpi. See *bits per inch*.

directory: A file used to locate files on a volume. A directory file contains a list of files and their unique internal identifications.

extent: One or more adjacent clusters allocated to a file or a portion of a file.

FDL: See *File Definition Language*

file: An organized collection of related items (records) maintained in an accessible storage area, such as disk or tape.

file header: A block in the index file describing a file on a Files-11 On-Disk Structure disk, including the location of the file's extents. There is at least one file header for every file on the disk.

File Definition Language: A special-purpose language used to write file creation and run-time specifications for data files. These specifications are written in text files called FDL files; they are then used by the VAX RMS utilities and library routines to create the actual data files.

file organization: The physical arrangement of data in the file. You select the specific organization from those offered by VAX RMS, based on your individual needs for efficient data storage and retrieval. See *indexed file organization*, *relative file organization*, and *sequential file organization*.

Files-11 On-Disk Structure: The standard physical disk structure used by VAX RMS.

fixed control area: A fixed-size area, prefixed to a variable-length record, containing additional information that can be processed separately and that may have no direct relationship to the other contents of the record. For example, the fixed control area might contain line sequence numbers for use in editing operations.

fixed-length record format: Property of a file in which all records are of the same size. This format provides simplicity in determining the exact location of a record in the file and eliminates the need to prefix a record size field to each record.

home block: A block in the index file, normally next to the bootstrap block, that identifies the volume as a Files-11 On-Disk Structure volume and provides specific information about the volume, such as volume label and protection.

index: The structure that allows retrieval of records in an indexed file by key value. See *key (indexed files)*.

index file: A file on each Files-11 On-Disk Structure volume that provides the means for identification and initial access to the volume. Contains the access data for all files (including itself) on the volume: bootstrap block, home block, file headers.

indexed file organization: A file organization that allows random retrieval of records by key values and sequential retrieval of records in sorted order by key value. See *key (indexed files)*.

interrecord gap: An interval of blank space between data records on the recording surface of a magnetic tape. Allows the tape unit to decelerate, and stop if necessary, after a record operation and then accelerate before the next record operation.

IRG: See *interrecord gap*.

key: *Indexed files:* a character string, a packed decimal number, a 2- or 4-byte unsigned binary number, or a 2- or 4-byte signed integer within each data record in an indexed file. You define the length and location within the records; VAX RMS uses the key to build an index. See *primary key*, *alternate key*, and *random access by key value*.

Relative files: The relative record number of each data record cell in a data file; VAX RMS uses the relative record numbers to identify and access data records in a relative file in random access mode. See *relative record number*.

locate mode: Technique used for a record input operation in which the data records are not copied from the VAX RMS I/O buffer, but a pointer is returned to the record in the VAX RMS I/O buffer. See *move mode*.

move mode: Technique used for a record transfer in which the data records are copied between the I/O buffer and your program buffer for calculations or operations on the record. See *locate mode*.

multiple-extent file: A disk file having two or more extents.

native mode: The processor's primary execution mode in which the programmed instructions are interpreted as byte-aligned, variable-length instructions that operate on the following data types: byte, word, longword, and quadword integers; floating and double floating character strings; packed decimals; and variable-length bit fields. The other instruction execution mode is compatibility mode.

primary key: The mandatory key within the data records of an indexed file; used by VAX RMS to determine the placement of records within the file and to build the primary index. See *key (indexed files)* and *alternate key*.

random access by key: *Indexed files only:* Retrieval of a data record in an indexed file by either a primary or alternate key within the data record. See *key (indexed files)*.

Relative files only: Retrieval of a data record in a relative file by the relative record number of the record. See *key (relative files)*.

random access by record's file address: Retrieval of a record by the record's unique address, which VAX RMS returns to you. This record access mode is the only means of randomly accessing a sequential file containing variable-length records.

random access by relative record number: Retrieval of a record by its relative record number. See *relative record number*. For relative files and sequential files (on disk devices) that contain fixed-length records, random access by relative record number is synonymous with random access by key. See *random access by key (relative files)*.

record: A set of related data that your program treats as a unit.

record access mode: The manner in which VAX RMS retrieves or stores records in a file. Available record access modes are determined by the file organization and specified by your program.

record access mode switching: Term applied to the switching from one type of record access mode to another while processing a file.

record blocking: The technique of grouping multiple records into a single block. On magnetic tape an IRG is placed after the block rather than after each record. This technique reduces the number of I/O transfers required to read or write the data; and, in addition (for magnetic tape), increases the amount of usable storage area. Record blocking also applies to disk files.

record cell: A fixed-length area in a relative file that can contain a record. The concept of fixed-length record cells lets VAX RMS directly calculate the record's actual position in the file.

record's file address: The unique address of a record in a file, which is returned by VAX RMS whenever a record is accessed, allows records in disk files to be accessed randomly regardless of file organization. This address is valid only for the life of the file. If an indexed file is reorganized, then the record's file address of each record will typically change.

relative file organization: The arrangement of records in a file where each record occupies a cell of equal length within a bucket. Each cell is assigned a successive number, called a relative record number, which represents the cell's position relative to the beginning of the file.

record format: The way a record physically appears on the recording surface of the storage medium. The record format defines the method for determining record length.

record length: The size of a record; that is, the number of bytes in a record.

record locking: A facility that prevents access to a record by more than one record stream or process until the initiating record stream or process releases the record.

Record Management Services: See VAX RMS (Record Management Services)

relative record number: An identification number used to specify the position of a record cell relative to the beginning of the file; used as the key during random access by key mode to relative files.

reorganization: A record by record copy of an indexed file to another indexed file with the same key attributes as the input file.

RFA: See *record's file address*.

RMS: See VAX RMS (*Record Management Services*)

RMS-11: A set of routines that are linked with compatibility mode and PDP-11 programs, and provide similar functional capabilities to VAX RMS. The file organizations and record formats used by RMS-11 are very similar to those of VAX RMS; one exception is that RMS-11 does not support Prolog 3 indexed files, which are supported by VAX RMS.

sequential file organization: The arrangement of records in a file in one-after-the-other fashion. Records appear in the order in which they were written.

sequential record access mode: Record storage or retrieval that starts at a designated point in the file and continues in one-after-the-other fashion through the file. That is, records are accessed in the order in which they physically appear in the file.

synchronous record operation: An operation in which your program does not regain control until after the completion of a record retrieval or storage request. See *asynchronous record operation*.

stream: An access window to a file associated with a record access control block, supporting record operation requests.

stream record format: Property of a file specifying that the data in the file is interpreted as a continuous sequence of bytes, without control information, except for certain terminator characters that are recognized as record separators. Stream record format applies to sequential files only.

track: A collection of blocks at a single radius on one recording surface of a disk.

variable-length record format: Property of a file in which records need not be of the same size.

variable with fixed-length control record format: Property of a file in which records of variable-length contain an additional fixed control area capable of storing data that may have no bearing on the other contents of the record. Variable with fixed-length control record format is not applicable to indexed files.

VAX RMS (Record Management Services): The file and record access subsystem of the VAX/VMS operating system. VAX RMS helps your application program process records within files, thereby allowing interaction between your application program and its data.

volume: *Disk:* An ordered set of 512-byte blocks. The basic medium that carries Files-11 On-Disk Structure files.

Magnetic tape: A reel of magnetic tape, which may contain a part of a file, a complete file, or more than one file.

volume set: A collection of related volumes.

Index

A

Access

- random • 3-16
 - with spatial locality • 3-17
 - with temporal locality • 3-17
- sequential • 3-16
- shared • 10-36
 - in a VAXcluster • 3-35
 - to process-permanent files • 6-23

Access category

- group • 4-27
- owner • 4-27
- system • 4-27
- world • 4-27

Access control list

- See ACL

Access mode

- See Record access mode

ACCESS primary

- BLOCK_IO attribute • 7-4
- DELETE attribute • 7-4
- GET attribute • 7-4
- PUT attribute • 7-4
- RECORD_IO attribute • 7-4
- TRUNCATE attribute • 7-4
- UPDATE attribute • 7-4

ACL (access control list) • 1-13

- ACL-based protection • 4-28

ADD command • 4-4, 10-33

AGAIN command • 10-14

Allocation • 3-27, 4-38, A-1

ALLOCATION attribute • 4-39

Allocation quantity option • 4-39

Alternate index • 3-23

Alternate key • 3-19

Alternate record structure • 10-26

ANALYSIS_OF_AREA primary attribute •

- 10-1, 10-28

ANALYSIS_OF_KEY primary attribute • 10-1,

- 10-28

Analysis section • 4-5, 10-1, 10-34

Analyze/RMS_File Utility (ANALYZE/RMS_

- FILE) • 1-16, 10-1, 10-34

examining prologue • 3-19

file optimizing • 4-5

with FDL files • 4-3

/ANALYZE qualifier • 10-34

ANL file type • 10-7

Application design • 2-1, 2-28

shared access consideration • 3-3

space consideration • 3-2

speed consideration • 3-1

Approximate key match • 8-14

Area • 3-27

multiple • 3-7, 3-28, 3-30

defining in an FDL file • 3-28

on a volume set • 3-27

AREA DESCRIPTOR structure • 10-22

AREA primary • 4-39

BEST_TRY_CONTIGUOUS attribute • 4-39

EXACT_POSITIONING attribute • 4-40

POSITION attribute

ANY_CYLINDER option • 4-39

CYLINDER option • 4-40

FILE_ID option • 4-40

FILE_NAME option • 4-40

LOGICAL option • 4-40

VIRTUAL option • 4-40

VOLUME attribute • 4-40

AREA primary attribute

ALLOCATION secondary attribute • 3-28

BEST_TRY_CONTIGUOUS secondary attribute • 3-27

CONTIGUOUS secondary attribute • 3-27

DATA_AREA secondary attribute • 3-29

INDEX_AREA secondary attribute • 3-29

LEVEL1_INDEX_AREA secondary attribute • 3-29

Areas option • 4-39

ASSIGN command • 4-18

/TRANSLATION_ATTRIBUTES qualifier • 5-8

Index

Asynchronous operation • 8-21, 8-23
performance • 9-10
Attribute • 4-3, 4-12

B

Backup Utility (BACKUP) • 10-2
eliminating extents • 9-10
making archive copies • 10-37
BEST_TRY_CONTIGUOUS attribute • 4-39
Bits per inch (bpi)
defined • 1-11
Block • 3-7
defined • 1-6, 2-2
input/output • 8-16 to 8-17
spanning • 3-12
BLOCK_IO attribute • 7-4
BLOCK_SPAN attribute • 3-12, 4-38
Block size option • 4-36
Bucket • 3-7, 3-21
boundary • 3-23
defined • 2-2
reclaiming • 3-20, 10-36
size
considering performance • 3-30
for indexed files • 7-24
for relative files • 7-23
relative to index depth • 3-29
split • 3-8, 3-26, 9-16, 10-37
BUCKET_SIZE attribute • 4-36, 7-23, 7-24
Buffer • 7-19 to 7-27
cache • 7-22, 7-25
defined • 7-6
global • 3-10, 3-33, 7-21, 7-25 to 7-27
performance • 9-11
with deferred write option • 3-33
I/O • 7-20
size • 3-2
key • 9-16, 9-18, 9-22
local • 3-33, 7-25
multiple • 3-9
number of • 3-13, 3-31, 3-32
record • 7-21, 9-23, 9-25
record header • 9-20, 9-23, 9-24
strategies • 7-20 to 7-22

Buffer (cont'd.)
user • 9-21
/BUFFER_COUNT qualifier • 7-23, 7-24,
7-25
Buffer area
requirement for Get service • 8-3
Buffered I/O byte count quota • 9-10
Byte
defined • 1-3

C

CARRIAGE_CONTROL attribute • 4-38
Cell
fixed length • 3-15
CELL AND RECORD structure • 10-20
/CHECK qualifier • 10-2
Check report • 10-2, 10-6
Cluster
defined • 1-6
Command
for Analyze/RMS_File Utility • 10-14
for Edit/FDL Utility • 4-4
Completion status value field • 5-16
CONNECT primary attribute
ASYNCHRONOUS secondary attribute •
9-10, 9-18, 9-22, 9-24, 9-25
DELETE_ON_CLOSE secondary attribute •
9-14
END_OF_FILE secondary attribute • 9-12
FAST_DELETE secondary attribute • 9-11
FAST_DELETE secondary attribute • 9-15,
9-25
FILL_BUCKETS secondary attribute •
9-16, 9-22
GLOBAL_BUFFER_COUNT secondary
attribute • 9-11
KEY_GREATER_EQUAL attribute • 8-11,
8-12
KEY_GREATER_EQUAL secondary
attribute • 9-15, 9-18
KEY_GREATER_THAN attribute • 8-11,
8-12
KEY_GREATER_THAN secondary
attribute • 9-15, 9-19

Index

CONNECT primary attribute (cont'd.)

- KEY_LIMIT secondary attribute • 9-16, 9-19
- KEY_OF_REFERENCE secondary attribute • 9-15, 9-19
- LOCATE_MODE secondary attribute • 9-11
- LOCATE_MODE secondary attribute • 9-19
- LOCK_ON_READ attribute • 7-13
- LOCK_ON_READ secondary attribute • 9-19
- LOCK_ON_WRITE attribute • 7-13
- LOCK_ON_WRITE secondary attribute • 9-20, 9-22
- MANUAL_LOCKING secondary attribute • 9-20
- MANUAL_UNLOCKING attribute • 7-13, 7-14
- MULTIBLOCK_COUNT attribute • 7-23
- MULTIBLOCK_COUNT secondary attribute • 3-13, 9-11
- MULTIBUFFER_COUNT attribute • 7-21, 7-23, 7-25
- MULTIBUFFER_COUNT secondary attribute • 3-13, 3-16, 3-31, 9-11
- NOLOCK attribute • 7-13
- NOLOCK secondary attribute • 9-18
- NONEXISTENT_RECORD attribute • 7-13, 8-11
- NONEXISTENT_RECORD secondary attribute • 9-19
- READ_AHEAD secondary attribute • 9-11, 9-20
- READ_REGARDLESS attribute • 7-14
- READ_REGARDLESS secondary attribute • 9-20
- TIMEOUT_PERIOD attribute • 7-14
- TIMEOUT_PERIOD secondary attribute • 9-21, 9-23
- TRUNCATE_ON_PUT secondary attribute • 9-13, 9-23
- UPDATE_IF attribute • 8-10
- UPDATE_IF secondary attribute • 9-13, 9-24
- WAIT_FOR_RECORD attribute • 7-14

CONNECT primary attribute (cont'd.)

- WAIT_FOR_RECORD secondary attribute • 9-21
- WRITE_BEHIND secondary attribute • 9-12, 9-24
- Connect service • 8-7
 - and asynchronous operations • 8-23
 - and next record • 8-19, 8-20
 - effect on next-record position • 8-20
- Contiguity • 10-35
- CONTIGUOUS attribute • 4-39
- Contiguous best try option • 4-39
- Contiguous option • 4-39
- CONTROL_FIELD_SIZE attribute • 4-37
- Convert/Reclaim Utility (CONVERT/RECLAIM)
 - 1-17, 3-19
 - with Prolog 3 files • 3-20, 10-36
- Convert routine
 - CONV\$CONVERT routine • 4-29
 - CONV\$PASS_FILES routine • 4-29
 - CONV\$PASS_OPTIONS routine • 4-29
- Convert Utility (CONVERT) • 1-17, 9-10
 - creating data files • 4-22, 4-23
 - making a file contiguous • 10-36
 - optimizing data files • 10-34
 - populating a file • 4-28
 - reorganizing files • 10-37
 - reorganizing noncontiguous files • 3-31, 10-36
 - with corrupted files • 10-1, 10-2
 - with FDL files • 4-3
 - with Prolog 1 and 2 files • 3-19
 - with Prolog 3 files • 3-20
- COPY/CONTIGUOUS command • 9-10
- COPY command
 - /CONTIGUOUS qualifier • 10-35
- CREATE_IF attribute • 4-35
- Create/FDL Utility (CREATE/FDL) • 1-18, 4-3, 4-22, 10-1
 - Create-if option • 4-22, 4-35, 5-12
 - /CREATE qualifier • 4-15
 - Create service • 4-22, 5-11, 5-12
 - for process-permanent files • 6-24
- Creation-time option • 3-10, 4-1, 4-3, 4-22, 4-35, 4-36

Index

Creation-time option (cont'd.)
 See also File opening option
CTRL/Z function • 4-5
Current context
 current-record position • 8-19 to 8-20
 listed for VAX RMS services • 8-18
 next-record position • 8-20 to 8-21
Current-record context • 8-18
Current-record position
 when adding sequential records • 8-4
 when updating records • 8-5
Cylinder
 boundary • 3-15
 defined • 1-8
 boundary option • 4-39
 position option • 4-39

D

Data
 compression • 3-20
DATA BUCKET structure • 10-20, 10-26
Data file
 creating • 4-22
 creating with FDL\$CREATE routine • 4-19, 4-23
 reorganizing • 10-35
Data reliability • 9-13
Data storage
 and file organization • 3-2
Data type
 nonstring • 3-20
 string • 3-20
Date information option • 4-36
DATE primary • 4-36
Default extension option • 4-40
DEFERRED_WRITE attribute • 7-24, 7-25
Deferred-write processing • 9-10
DEFINE command • 4-18, 6-16
 /TRANSLATION_ATTRIBUTES qualifier • 5-8
DELETE attribute
 of ACCESS primary • 7-4
 of SHARING primary • 7-5
DELETE command • 4-4, 10-33

Delete service • 8-2, 8-6
 high-level language equivalents • 8-2
 run-time options • 9-25
Design graphics mode • 4-15
Design mnemonic • 4-18
Directory
 master file • 6-14
 reference
 absolute • 6-14
 relative • 6-14
DIRECTORY_ENTRY attribute • 4-36
Directory specification
 normal • 6-14 to 6-16
 rooted • 6-16 to 6-22
Directory tree
 defined • 6-14
Disconnect service • 8-7
Disk block • 3-7
Disk cylinder • 3-7
Disk quota • 3-6
Disk volume • 3-7
DOWN command • 10-14, 10-16
DUMP command • 10-14

E

EDF\$MAKE_FDL logical name • 4-18
EDIT/ACL command • 4-28
Edit/FDL Utility (EDIT/FDL) • 1-19
 calculating bucket size • 3-15, 3-16, 3-30
 calculating extension size • 3-6, 9-9
 commands • 4-4
 contiguous files • 3-5
 creating areas for index structures • 3-28
 creating FDL files • 4-3, 4-6
 default value • 4-15
 invoking a script • 4-6
 optimization algorithms • A-1
 Optimize script • 10-1, 10-28
 prompt • 4-15
 specifying run-time attributes • 9-2 to 9-5
Equal-or-next search option • 8-11, 8-12
Equivalence string
 defined • 6-4
Erase service • 5-11

Error

- check • 10-2
- signaling • 5-16
- software • 10-2

EXACT_POSITIONING attribute • 4-40

Exact key match • 8-14

EXIT command • 4-4, 10-14

Expanded string • 6-4 to 6-5
defined • 6-4

/EXTEND_QUANTITY qualifier • 9-9

Extended attribute block

See XAB

EXTENSION attribute • 4-40

Extension size • A-1

- calculating • 9-9
- performance • 9-9, 9-10

Extent

- defined • 1-7, 9-9

F

FAB (file access block) • 1-15, 4-2

FAB\$B_BKS field • 3-29, 4-36, 7-23, 7-24

FAB\$B_BLS field • 4-36

FAB\$B_DEQ field • 9-9

FAB\$B_DNS field • 9-8

FAB\$B_FAC field • 9-7

- FAB\$V_BIO option • 7-4
- FAB\$V_BRO option • 7-4
- FAB\$V_DEL option • 7-4
- FAB\$V_GET option • 7-4
- FAB\$V_PUT option • 7-4
- FAB\$V_TRN option • 7-4
- FAB\$V_UPD option • 7-4

FAB\$B_FNS field • 6-6, 9-8

FAB\$B_FSZ field • 4-37

FAB\$B_ORG field • 4-36

FAB\$B_RAT field • 4-38

FAB\$B_RFM field • 4-38

FAB\$B_RTV field • 9-10, 9-11

FAB\$B_SHR field • 9-7

- FAB\$V_MSE option • 7-5, 7-27
- FAB\$V_NIL option • 7-5
- FAB\$V_SHRDEL option • 7-5
- FAB\$V_SHRGET option • 7-5, 7-27

FAB\$B_SHR field (cont'd.)

FAB\$V_SHRPUT option • 7-5

FAB\$V_SHRUPD option • 7-5

FAB\$V_UPI option • 7-9

FAB\$V_UPI option • 7-5

FAB\$L_ALQ field • 4-39

FAB\$L_DNA field • 6-4, 9-8

FAB\$L_FNA field • 6-4, 6-6, 9-8

FAB\$L_FOP field • 4-35

FAB\$V_CBT option • 4-39

FAB\$V_CTG option • 4-39

FAB\$V_DFW option • 3-17, 3-18, 3-32,
3-33, 7-24, 7-25, 9-10, 9-22

FAB\$V_DLT option • 9-14

FAB\$V_MXV option • 4-35

FAB\$V_NAM option • 6-5

FAB\$V_NEF option • 8-20

FAB\$V_NEF option • 8-19, 9-17

FAB\$V_NFS option • 9-17

FAB\$V_OFF option • 6-10, 6-11, 6-12

FAB\$V_POS option • 9-17

FAB\$V_PPF option • 6-23

FAB\$V_RCK option • 9-14

FAB\$V_RWC option • 9-17

FAB\$V_RWO option • 9-17

FAB\$V_SCF option • 9-14

FAB\$V_SPL option • 9-14

FAB\$V_SQO option • 9-12

FAB\$V_SUP option • 4-35

FAB\$V_TMD option • 4-35

FAB\$V_TMP option • 4-36

FAB\$V_UFO option • 7-5, 9-17

FAB\$V_WCK option • 9-14

FAB\$L_MRN field • 4-37

FAB\$L_MRS field • 4-37

FAB\$L_NAM field • 6-10, 9-8

FAB\$L_STV field • 9-17

FAB\$W_DEQ field • 4-40, 9-10

FAB\$W_GBC field • 7-21, 7-27, 9-11

\$FABDEF

for defining symbols to USEROPEN routine
• 5-13

Fast-delete option • 8-6, 9-11

FDL (File Definition Language) • 3-16, 4-3
attributes • 4-3

Index

FDL (File Definition Language) (cont'd.)

- defined • 1-14

- scripts • 4-3

- syntax • 4-3

FDL\$PARSE routine

- for supplying predefined FDL attributes • 9-1

FDL Editor

- See Edit/FDL Utility (EDIT/FDL)

FDL file

- creating • 4-3

- creating data files • 4-22

- creating with FDL\$GENERATE routine • 4-19

- designing • 4-15

- examining with ANALYZE/RMS_FILE • 10-1

- generating from a data file • 10-28

- /FDL qualifier • 10-28

FDL routine

- FDL\$CREATE routine • 4-19, 4-23, 6-3

- FDL\$GENERATE routine • 4-19

- FDL\$PARSE routine • 4-19, 6-3, 9-2

- example • 9-26 to 9-28

- FDL\$RELEASE routine • 4-19, 6-3, 9-2

- example • 9-26 to 9-28

Field

- defined • 1-3

File

- See also File characteristic

- access in a VAXcluster • 3-35

- aligning • 3-15

- concepts • 1-2

- contiguity • 3-5, 3-28

- corruption • 10-2

- defined • 1-3

- extension • 3-27

- extension size • 3-5

- FDL • 4-3, 4-22, 10-1, 10-28

- header • 3-11, 3-15, 3-19, 10-14

- indexed • 10-33, 10-36

- initial allocation • 3-5

- internal structure • 10-1

- locking in a VAXcluster • 3-35

- on magnetic tape • 1-12

File (cont'd.)

- protection • 4-26

- sharing • 3-3

- specifying one or many • 5-21

- structure • 10-2, 10-14

File access

- category summary • 4-27

- control • 4-27

- defaults • 7-6

- delete • 4-27

- execute • 4-27

- read • 4-27

- write • 4-27

File access block

- See FAB

- FILE ATTRIBUTES structure • 10-16, 10-20, 10-22

- File characteristic • 4-18, 4-35, 4-36

File design

- attributes • 3-4

- File disposition • 9-14

- File header • 1-10

- FILE HEADER structure • 10-16, 10-20, 10-22

File opening option

- See also Creation-time options

- adding records • 9-12 to 9-13

- data reliability • 9-13

- file access and sharing • 9-7

- file disposition • 9-14

- file performance • 9-8 to 9-12

- file specification • 9-7 to 9-8

- for indexed files • 9-15 to 9-16

- for magnetic tape processing • 9-16 to 9-17

- for nonstandard file processing • 9-17

- record access • 9-12

File organization • 2-16

- See Indexed file

- See Relative file

- See Sequential file

- defined • 1-3

- selecting • 2-1

- File organization option • 4-36

- File positioning • 4-38

FILE primary
 TEMPORARY attribute • 4-35

FILE primary attribute
 ALLOCATION secondary attribute • 3-5, 3-28
 BEST_TRY_CONTIGUOUS secondary attribute • 3-5
 BUCKET_SIZE secondary attribute • 3-15, 3-16, 3-29
 CONTIGUOUS secondary attribute • 3-5, 3-28
 DEFAULT_NAME secondary attribute • 6-4, 9-8
 DEFERRED_WRITE secondary attribute • 3-17, 3-32, 9-10, 9-22
 EXTENSION secondary attribute • 3-6, 9-9, 9-10
 GLOBAL_BUFFER_COUNT secondary attribute • 3-10
 MT_CLOSE_REWIND secondary attribute • 9-17
 MT_CURRENT_POSITION secondary attribute • 9-17
 MT_NOT_EOF secondary attribute • 9-17
 MT_OPEN_REWIND secondary attribute • 9-17
 NAME secondary attribute • 6-4, 9-8
 NON_FILE_STRUCTURED secondary attribute • 9-17
 PRINT_ON_CLOSE secondary attribute • 9-14
 READ_CHECK secondary attribute • 9-14
 SEQUENTIAL_ONLY secondary attribute • 9-12
 SUBMIT_ON_CLOSE secondary attribute • 9-14
 USER_FILE_OPEN attribute • 7-5
 USER_FILE_OPEN secondary attribute • 9-17
 WINDOW_SIZE secondary attribute • 9-10, 9-11
 WRITE_CHECK secondary attribute • 9-14

File primary attribute
 ALLOCATION attribute • 4-39
 BEST_TRY_CONTIGUOUS attribute • 4-39

File primary attribute (cont'd.)
 BUCKET_SIZE attribute • 4-36, 7-23, 7-24
 CONTIGUOUS attribute • 4-39
 CONTROL_FIELD_SIZE attribute • 4-37
 CREATE_IF attribute • 4-35
 DEFERRED_WRITE attribute • 7-24, 7-25
 DIRECTORY_ENTRY attribute • 4-36
 EXTENSION attribute • 4-40
 GLOBAL_BUFFER_COUNT attribute • 7-21, 7-27
 MAX_RECORD_NUMBER attribute • 4-37
 MAXIMIZE_VERSION attribute • 4-35
 MT_BLOCK_SIZE attribute • 4-36
 MT_PROTECTION attribute • 4-37
 ORGANIZATION attribute • 4-36
 OWNER attribute • 4-37
 PROTECTION attribute • 4-37
 REVISION attribute • 4-36
 SUPERSEDE attribute • 4-35

File processing
 many files • 5-20 to 5-21
 nonstandard • 9-17
 single file • 5-19 to 5-20

File protection option • 4-36

Files—11 On-Disk Structure • 1-5
 file headers • 1-10
 home block • 1-10
 index file • 1-10

File sharing • 3-10, 9-7
 compatibility with subsequent record access • 7-7 to 7-8
 defaults • 7-6
 interlocked interprocess • 7-3, 7-7 to 7-8
 multistreaming • 7-3, 7-5
 no-access function • 7-5
 options • 7-5
 programming techniques • 7-17 to 7-18
 user-interlocked interprocess • 7-3, 7-5, 7-9

File specification • 6-3
 applicable services and routines • 5-10 to 5-18
 components • 5-1 to 5-3
 default • 5-5, 6-1 to 6-4, 9-8
 directory • 6-14 to 6-22

Index

File specification (cont'd.)

- format • 5-1 to 5-5, 6-6 to 6-7
 - for remote file access • 5-3 to 5-5, 5-9, 5-10
 - input • 6-11
 - maximum length • 5-2
 - output • 6-12
 - parsing • 5-9 to 5-11, 6-4 to 6-14
 - preprocessing • 5-10
 - primary • 5-5, 6-1 to 6-4, 9-8
 - process default • 5-5
 - program supplied • 5-5
 - program-supplied • 6-1 to 6-4
 - related • 5-5, 6-1 to 6-4, 6-10 to 6-11, 9-8
 - using • 5-1
 - using defaults • 5-5 to 5-6, 6-1 to 6-4
 - using logical name • 6-6 to 6-7
 - using name block • 5-10
 - using search lists • 5-10 to 5-21, 6-8 to 6-10
 - using SYS\$DISK • 6-2
 - using wildcard characters • 5-10 to 5-21
 - VAX/VMS Version 3 compatibility • 5-3, 5-5, 5-9
- File tuning
- See Tuning files
- File type
- ANL • 10-7
- Fill factor • 3-31
- Find service • 8-1, 8-3 to 8-4
- and key matches • 8-13
 - compared with Get service • 8-3
 - effect on next-record position • 8-21
 - high-level language equivalents • 8-1
 - improved performance • 8-4
 - requirement for end-of-file test • 8-4
 - run-time options • 9-18 to 9-21
- FIRST command • 10-14
- Fixed control area • 3-15
- Fixed control size option • 4-37
- Flush service • 7-9, 8-7
- FORMAT attribute • 4-38
- Free service • 7-12, 8-7

G

- GBLPAGES system parameter • 1-21
- GBLPAGFIL system parameter • 1-21
- GBLSECTIONS system parameter • 1-21
- Generic key match • 8-14
- GET attribute
 - of SHARING primary • 7-5, 7-27
- Get service • 8-1, 8-3
 - and current-record • 8-20
 - compared with Find service • 8-3
 - effect on next-record position • 8-21
 - high-level language equivalents • 8-2
 - requirement for end-of-file test • 8-4
 - run-time options • 9-18 to 9-21
- GLOBAL_BUFFER_COUNT attribute • 7-21, 7-27
- /GLOBAL_BUFFERS qualifier • 7-27
- Global buffer • 1-21
 - maximum number • 1-21
 - number • 7-21
 - restricted use • 7-26
 - with indexed file • 7-25
 - with relative file • 7-25
 - with shared file • 7-25 to 7-27
 - with shared sequential file • 3-14
- Global buffer count
 - example of run-time specification • 5-13 to 5-16
- Global page-file section • 1-21
- Global page table • 1-21
- Global section • 1-21
- Glossary information • GLOSS-1

H

- Hard positioning option • 4-40
- Hardware error • 10-2
- HELP command • 4-4, 10-14
- Home block • 1-10

I

- I/O and performance • 3-1

Index

I/O unit • 3-7, 3-9, 3-13

Index

structure • 3-19

INDEX BUCKET structure • 10-26

Index depth • A-3

Indexed file • 2-22, 3-19

advantages and disadvantages • 2-26

allocating • A-1

alternate key • 2-24

bucket size • 3-7, 3-29, 7-24, A-2

buffering • 7-24 to 7-25

compression • 3-3, 3-19

deferred write • 3-10

designing • 3-19 to 3-34

examining • 10-22

fill factor • 3-8

global buffers • 7-25

key type • 2-24

making contiguous • 10-36

optimizing • 3-19 to 3-34

primary key • 2-24

Prolog 1 and 2 • 3-19

reclaiming buckets in • 10-36

record access • 8-12 to 8-16, 8-16

redesigning • 10-33

reorganizing • 10-37

run-time options • 9-15 to 9-16

tuning • 3-19 to 3-34

with global buffers • 3-33

Indexed file organization

defined • 1-4

/INDEXED qualifier • 7-25

Index structure • 3-29

Level 0 • 3-21

Level 1 • 3-21

primary • 3-21

INITIALIZE command

and window size • 9-10

/INTERACTIVE qualifier • 10-14

Internal buffer

for storing records • 8-3

Interrecord gap

See IRG

INVOKE command • 4-4, 4-6

IRG • 1-11

Item

defined • 1-3

K

Key

alternate

duplicate values • 3-27

performance of • 3-26

buffer • 9-16, 9-18, 9-22

compression

front • 3-20

rear • 3-20

duplicate values • 2-25

for Prolog 1 and 2 files • 3-19

null • 2-25

null value • 3-23

number of • 3-27

primary • 3-20, 3-26

segmented • 3-20

size • 9-16, 9-19, 9-22

used to store indexed records sequentially
• 2-7

Key 0 • 3-21

KEY_GREATER_EQUAL attribute • 8-11,
8-12

KEY_GREATER_THAN attribute • 8-11, 8-12

Key buffer • 8-4

Key characteristics option • 4-37

KEY DESCRIPTOR structure • 10-22

Key-greater-than-or-equal search option

See also Equal-or-next search option

Key-greater-than search option

See also Next search option

Key match • 8-14 to 8-16

approximate • 8-14

exact • 8-14

generic • 8-13, 8-14

generic and approximate • 8-15 to 8-16

Key of reference • 2-6

KEY primary attribute • 4-37

DATA_AREA secondary attribute • 3-29

DATA_FILL secondary attribute • 3-31

INDEX_AREA secondary attribute • 3-29

INDEX_FILL secondary attribute • 3-31

Index

KEY primary attribute (cont'd.)

LEVEL1_INDEX_AREA secondary
attribute • 3-29

TYPE secondary attribute • 3-26

L

Level

number of • A-3

LIB\$FIND_FILE routine • 5-10 to 5-16

Line_Plot graph • 4-15, A-3

Locate mode

and record retrieval • 8-3

Lock

root • 3-35

LOCK_ON_READ attribute • 7-13

LOCK_ON_WRITE attribute • 7-13

Logical block position option • 4-40

Logical name

advantages • 5-6

concealed • 6-17

concealed attribute • 5-8

example program • 5-7

parsing • 5-9

rooted-device • 6-17

search list • 5-9, 6-8 to 6-10

translation of • 5-8, 6-6 to 6-7

types • 5-7 to 5-9

M

Magnetic tape processing

run-time options • 9-16 to 9-17

MANUAL_UNLOCKING attribute • 7-13, 7-14

Master file directory (MFD) • 6-14

MAX_RECORD_NUMBER attribute • 4-37

MAXIMIZE_VERSION attribute • 4-35

Maximize version option • 4-35

Maximum record number option • 4-37

Maximum record size

indexed file

Prolog 1 • 3-26

Prolog 2 • 3-26

Prolog 3 • 3-26

Maximum record size option • 4-37

Memory

nonpaged system dynamic • 9-10

releasing with the FDL\$RELEASE routine •
4-19

Memory cache • 3-14, 3-18

MFD

See Master file directory

Mode

interactive • 10-14

locate

performance • 9-11

MODIFY command • 4-5, 10-33

Edit/FDL Utility • A-2

MOUNT command

and window size • 9-10

MT_BLOCK_SIZE attribute • 4-36

MT_PROTECTION attribute • 4-37

Multiblock • 3-7, 3-13

defined • 2-2

MULTIBLOCK_COUNT attribute • 7-23

MULTIBUFFER_COUNT attribute • 7-21, 7-23

and record access type • 7-25

for sequential file • 7-23

Multibuffer count • 3-13, 3-16, 3-31, 3-32

Multiple area

defining • 3-27 to 3-29

Multiple service

for retrieving records • 8-4

MULTISTREAM attribute • 7-5

N

NAM (name) block

address field • 5-11

and resulting file specification • 5-10

and Search service • 5-10

presence of a search list • 5-11

presence of a wildcard character • 5-11

support by FDL • 5-12

support by languages • 5-12

using • 5-16 to 5-18

NAM\$B_RSS field • 6-10

NAM\$_ESA field • 6-4

NAM\$_RLF field • 6-4, 6-10, 9-8

NAM\$_RSA field • 6-4, 6-10

NAM\$_DVI field • 6-5
 NAM\$_DID field • 6-5
 NAM\$_FID field • 6-5
 \$NAMDEF
 for defining symbols to USEROPEN routine
 • 5-13
 NEXT command • 10-14, 10-16, 10-20
 Next-record position • 8-20 to 8-21
 use with sequential access • 8-20
 Next search option • 8-11, 8-12
 Next volume service • 8-7
 Node
 lock-mastering • 3-35
 lock-requesting • 3-35
 /NOINTERACTIVE qualifier • 10-34
 NOLOCK attribute • 7-13
 NONEXISTENT_RECORD attribute • 7-13,
 8-11
 Nonstandard file processing
 run-time options • 9-17
 Normal directory syntax • 6-14 to 6-16

O

Open-by-name-block option • 5-11, 6-5
 and performance • 6-7
 Open service • 5-11
 for process-permanent files • 6-24
 Optimization
 Edit/FDL Utility • A-1
 of indexed file • 10-34
 Organization
 See File organization
 ORGANIZATION attribute • 4-36
 OWNER attribute • 4-37

P

Parity bit • 1-11
 Parse service • 5-10 to 5-16
 Performance • 3-1, 9-8 to 9-12
 and asynchronous processing • 9-10
 and extension size • 9-9
 and fast-delete option • 9-11

Performance (cont'd.)

 and global buffer count • 9-11
 and locate mode • 9-11
 and record locking • 7-2
 and window size • 9-9 to 9-10
 buffers • 9-11
 deferred-write option • 3-34, 9-10
 effect of compression • 3-20
 extension size • 9-10
 I/O in VAXcluster • 3-36
 in a VAXcluster • 3-34
 multiblock count • 9-11
 read-ahead option • 9-11
 recommendations for a VAXcluster • 3-36
 sequential access • 9-12
 using Prolog 3 • 3-20
 window size • 9-11
 write-behind option • 9-12
 Pointer
 retrieval • 9-9
 POSITION attribute
 ANY_CYLINDER option • 4-39
 CYLINDER option • 4-40
 FILE_ID option • 4-40
 FILE_NAME option • 4-40
 LOGICAL option • 4-40
 VIRTUAL option • 4-40
 Primary attribute • 4-12
 Primary record structure • 10-26
 Process
 asynchronous system trap limit (ASTLM)
 resource • 1-21
 buffered I/O limit (BIOLM) resource • 1-21
 direct I/O limit (DIOLM) resource • 1-21
 open file limit (FILLM) resource • 1-22
 record-locking quota (ENQLM) resource •
 1-22
 resources • 1-20
 Process default • 4-18
 Process I/O segment • 1-21
 Process-permanent file • 1-21, 6-23
 access to • 6-23
 implications for indirect access • 6-24
 PROHIBIT attribute • 7-5
 Prolog • 3-24

Index

Prolog 1 • 3-19
Prolog 2 • 3-19
Prolog 3 • 3-20, 10-36
PROLOG structure • 10-20, 10-22
Prologue • 3-15, 3-19
Protection
 access category • 4-27
 ACL-based • 1-13, 4-28
 disk and tape volumes • 1-13
 UIC-based • 1-13, 4-27
PROTECTION attribute • 4-37
PUT attribute
 of ACCESS primary • 7-4
 of SHARING primary • 7-5
Put service • 8-2, 8-4 to 8-5
 and next record • 8-21
 effect on next-record position • 8-21
 high-level language equivalents • 8-2
 run-time options • 9-21 to 9-24

Q

Queue I/O Request system service • 7-5, 9-17
QUIT command • 4-5

R

RAB (record access block) • 1-15
RAB\$_KRF field • 9-15, 9-19
RAB\$_KSZ field • 8-10, 8-11, 8-16, 9-16, 9-19, 9-22
RAB\$_MBC field • 3-13, 7-23, 9-11
RAB\$_MBF field • 3-13, 3-31, 7-21, 7-23, 7-25, 9-11
RAB\$_RAC field
 RAB\$_KEY option • 8-8, 9-12, 9-20, 9-22
 RAB\$_RFA option • 8-8, 9-12, 9-20, 9-22
 RAB\$_SEQ option • 8-8, 9-12, 9-20, 9-22
RAB\$_TMO field • 7-14, 7-17, 7-18, 9-21
RAB\$_KBF field • 8-10, 8-11, 8-16, 9-16, 9-18, 9-22

RAB\$_RBF field • 9-23, 9-25
RAB\$_RBZ field • 9-23, 9-25
RAB\$_RHB field • 9-20, 9-23, 9-24
RAB\$_ROP field • 9-7
 RAB\$_ASY option • 8-22, 8-23, 9-10, 9-18, 9-22, 9-24, 9-25
 RAB\$_EOF option • 8-18, 8-20, 9-12
 RAB\$_EQNXT option • 9-15, 9-18
 RAB\$_FDL option • 9-11, 9-15, 9-25
 RAB\$_KGE option • 8-11, 8-12
 RAB\$_KGT option • 8-11, 8-12
 RAB\$_LIM option • 9-16, 9-19
 RAB\$_LOA option • 9-16, 9-22
 RAB\$_LOC option • 9-11, 9-19
 RAB\$_NLK option • 9-18
 RAB\$_NLK option • 7-14
 RAB\$_NXR option • 8-11, 9-19
 RAB\$_NXR option • 7-14
 RAB\$_NXT option • 9-15, 9-19
 RAB\$_RAH option • 3-14, 9-11, 9-20
 RAB\$_REA option • 9-19
 RAB\$_REA option • 7-14
 RAB\$_RLK option • 9-20, 9-22
 RAB\$_RLK option • 7-14
 RAB\$_RRL option • 9-20
 RAB\$_RRL option • 7-14
 RAB\$_TMO option • 9-21, 9-23
 RAB\$_TMO option • 7-14, 7-17, 7-18
 RAB\$_TPT option • 9-13, 9-23
 RAB\$_UIF option • 8-6, 8-10, 9-13, 9-24
 RAB\$_ULK option • 9-20
 RAB\$_ULK option • 7-14, 7-18
 RAB\$_WAT option • 9-21
 RAB\$_WAT option • 7-14, 7-18
 RAB\$_WBH option • 3-14, 9-12, 9-24
RAB\$_UBF field • 9-21
RAB\$_USZ field • 9-21
RAB\$_W_RBF • 8-3
RAB\$_W_RFA field • 8-16, 8-20, 9-20
RAB\$_W_RSZ • 8-3
\$RABDEF
 for defining symbols to USEROPEN routine • 5-13

Index

Random access

- by key value • 2-7 to 2-9, 8-8, 8-14 to 8-16
- by relative record number • 2-7 to 2-9, 8-8, 8-10, 8-11
- by RFA (record file address) • 2-9, 8-8, 8-16
- to indexed files • 2-8, 8-14 to 8-16, 8-16
- to relative files • 2-7, 8-11, 8-16
- to sequential files • 2-7, 8-10, 8-16
- with multibuffer count • 3-31

Random access mode

- defined • 1-4

READ_REGARDLESS attribute • 7-14

Record

- adding • 9-12 to 9-13
- blocking • 1-11
- contents • 2-2
- defined • 1-3
- deleting • 8-6, 9-25
- fixed format • 1-4, 3-11, 3-15
- fixed length format • 2-11
- fixed-length format • 2-12
- format • 2-9
- inserting • 8-4 to 8-5, 9-21 to 9-24
- locating • 8-3 to 8-4
- retrieving • 8-3 to 8-4, 9-18 to 9-21
- STREAM_CR format • 3-12
- STREAM_LF format • 3-12
- stream format • 1-4, 2-15, 3-11
- undefined format • 3-11, 3-12
- updating • 8-5 to 8-6, 9-24 to 9-25
- variable format • 1-4, 3-11, 3-12, 3-15
- variable format with fixed-length control
 - See VFC
- variable length format • 2-11
- variable-length format • 2-12
- VFC format • 1-4, 3-15

RECORD_IO attribute • 7-4

Record access • 9-7, 9-12

- options • 7-4
- stream context • 8-18

Record access block

- See RAB

Record access mode • 2-2

Record access mode (cont'd.)

- defined • 1-4
- for indexed files • 8-12 to 8-16
- for relative files • 8-10 to 8-12
- for sequential files • 8-9 to 8-10
- sequential • 2-2, 8-7, 8-11, 8-13
- specifying • 8-8 to 8-9, 9-12, 9-20, 9-22

Record attributes option • 4-37

Record buffer • 9-23, 9-25

- size • 9-23, 9-25

Record buffering

- See Buffering

Record file address

- See RFA

Record format • 1-3, 3-15

- defined • 1-4
- fixed • 3-23
- selecting • 2-1
- variable • 3-23

Record format option • 4-38

Record header buffer • 9-20, 9-23, 9-24

Record locking • 7-2 to 7-18, 9-7

- automatic • 7-11
- deadlock • 7-18
- Free service • 7-12
- manual unlocking • 7-13 to 7-18
- options • 7-13 to 7-18
- Release service • 7-12
- use with update operation • 8-4

Record operation • 8-1 to 8-7

RECORD primary attribute

- BLOCK_SPAN attribute • 4-38
- BLOCK_SPAN secondary attribute • 3-12
- CARRIAGE_CONTROL attribute • 4-38
- FORMAT attribute • 4-38
- SIZE attribute • 4-37

Record processing run-time option

- record deletion • 9-25
- record insertion • 9-21 to 9-24
- record retrieval • 9-18 to 9-21
- record update • 9-24 to 9-25

Record-processing service

- Connect • 8-7
- Disconnect • 8-7
- Flush • 8-7

Index

Record-processing service (cont'd.)

- Free • 8-7
- Next Volume • 8-7
- Release • 8-7
- Rewind • 8-7
- Truncate • 8-7
- Wait • 8-7
- Record reference vector
 - see RRV
- Record stream • 2-3
 - connecting to a file • 7-3 to 7-4
 - defined • 7-2
- Record stream connection option
 - See File opening options
- Record transfer mode
 - locate • 7-19
 - move • 7-19
- Related file position option • 4-40
- Relative file • 2-20, 3-15
 - advantages and disadvantages • 2-22
 - allocating • A-1
 - bucket size • 3-7, 3-15, 3-16, 7-23, A-2
 - buffering • 7-23 to 7-24
 - deferred write • 3-10
 - designing • 3-15 to 3-18
 - examining • 10-20, 10-21
 - maximum record size • 3-15
 - optimizing • 3-15 to 3-18
 - record access • 8-10 to 8-12, 8-16
 - tuning • 3-15 to 3-18
 - with global buffers • 3-18
- Relative file organization
 - defined • 1-4
- /RELATIVE qualifier • 7-24
- Relative record number • 1-4, 3-15
- Release service • 7-12, 8-7
- Remote file access
 - See also File specification
 - FORTTRAN program example • 5-7
- Rename service • 5-11
- Repeating characters
 - in compression • 3-20
- REST command • 10-16, 10-20
- Retrieval pointer • 9-9

RETURN key

- interactive mode • 10-14
- REVISION attribute • 4-36
- Revision data • 9-13
- Rewind service • 8-7
 - effect on next-record position • 8-21
- RFA
 - use of table for rapid access • 8-4
- RFA (record's file address) • 3-19, 8-16, 9-20
 - defined • 1-4
- RFA (record file address) • 10-37
 - access • 10-36
- RMS
 - See VAX RMS (Record Management Services)
- RMS_GBLBUFQUO system parameter • 1-21
- Rooted device logical name • 6-17
- Rooted directory logical name
 - for additional nesting • 6-20
- Rooted directory specification
 - concatenated • 6-19 to 6-21
 - syntax • 6-16 to 6-22
- Root level • 3-21
- Rotational latency
 - defined • 1-8
- RRV (record reference vector) • 3-8, 3-26
- Run-time option
 - example • 9-26 to 9-28
 - specifying • 9-1 to 9-6

S

Save set • 10-37

Script

- ADD_KEY • 4-6
- DELETE_KEY • 4-6
- INDEXED • 4-6
- invoking • 4-6
- OPTIMIZE • 4-6
- Optimize • 10-1
- RELATIVE • 4-6
- SEQUENTIAL • 4-6
- TOUCHUP • 4-6
- Touch-up • 10-33

Index

/SCRIPT=OPTIMIZE qualifier • 10-34

/SCRIPT qualifier • 10-34

Search list

See also File specification

and multiple file locations • 5-9, 5-10

defined • 5-9

example • 5-19, 5-20

translation • 6-8 to 6-10

Search service • 5-10 to 5-16

Secondary attribute • 4-12

Secondary completion status value field • 5-16

Secondary index data record

See SIDR

Secondary service

effect on next-record position • 8-21

Sector

defined • 1-8

Seek time

defined • 1-8

Sequential access • 8-7, 8-8

to indexed files • 2-6, 8-13

to relative files • 2-5, 8-11

to sequential files • 2-4

use with sequential files • 8-9

with multibuffer count • 3-31

Sequential access mode

defined • 1-4

Sequential file • 2-18

advantages and disadvantages • 2-19

allocating • A-1

buffering • 7-23

designing • 3-11 to 3-14

examining • 10-15, 10-17

maximum record size • 3-11

optimizing • 3-11 to 3-14

read-ahead and write-behind • 3-11

record access • 8-9 to 8-10, 8-16

tuning • 3-11 to 3-14

Sequential file organization

defined • 1-4

/SEQUENTIAL qualifier • 7-23

SET command • 4-5

SET DEFAULT command • 6-16

SET DEFAULT command (cont'd.)

/TRANSLATION_ATTRIBUTES qualifier • 6-17

SET FILE command • 1-22

/ACL qualifier • 4-28

/EXTENSION qualifier • 3-6

/GLOBAL_BUFFERS qualifier • 3-10, 7-27

SET PROTECTION command • 4-27

SET RMS_DEFAULT command • 1-22

/BLOCK_COUNT qualifier • 3-13

/BUFFER_COUNT qualifier • 3-9, 3-13, 3-16, 3-32, 7-23, 7-24, 7-25

/EXTEND_QUANTITY qualifier • 3-6, 9-9

/INDEXED qualifier • 7-25

/RELATIVE/BUFFER_COUNT qualifier • 3-17

/RELATIVE qualifier • 7-24

/SEQUENTIAL qualifier • 7-23

Shared access • 3-3

Shared file

see File sharing

SHARING primary

DELETE attribute • 7-5

GET attribute • 7-5, 7-27

MULTISTREAM attribute • 7-5

PROHIBIT attribute • 7-5

PUT attribute • 7-5

UPDATE attribute • 7-5

USER_INTERLOCK attribute • 7-9

USER_INTERLOCK attribute • 7-5

SHOW RMS_DEFAULT command • 3-9, 3-17, 3-32

current default extension size • 9-9

current process-default buffer count • 7-23 to 7-25

SIDR (secondary index data record) • 3-19, 3-23, 10-26

SIZE attribute • 4-37

Software error • 10-2

Speed

See Performance

Spool on close option • 9-14

SPR (Software Performance Report) • 10-2

Index

/STATISTICS qualifier • 10-7
Statistics report • 10-7, 10-13
Sticky default
 defined • 6-10
\$STOP routine • 5-16
STR\$GET1_DX routine • 5-13
SUPERSEDE attribute • 4-35
Supersede option • 4-35, 5-12
Surface_Plot graph • 4-16, A-3
Synchronous operation • 8-21, 8-22
SYS\$DISK
 applied to file specification • 6-2
SYS\$FILESCAN • 5-10
SYS\$OUTPUT
 for check report • 10-2
SYS\$SETDDIR service • 6-16
System
 default • 4-18
 resources • 1-20
System management • 3-9
System parameter • 1-20

T

Tape processing
 run-time options • 9-16 to 9-17
TEMPORARY attribute • 4-35
Temporary option • 4-36
 delete option • 4-35
Terminator
 CR • 3-12
 CR LF • 3-12
 FF • 3-12
 LF • 3-12
 VT • 3-12
Terminator variation • 3-12
TIMEOUT_PERIOD attribute • 7-14
TOP command • 10-15
Track
 defined • 1-8
 size • 3-16
/TRANSLATION_ATTRIBUTES qualifier
 • 5-8, 6-17
Tree structure • 10-14
 of indexed file • 10-22

Tree structure (cont'd.)
 of relative file • 10-20
 of sequential file • 10-15
TRUNCATE attribute • 7-4
Truncate-on-put option
 access requirement • 7-9
Truncate service • 8-6, 8-7
 effect on next-record position • 8-21
Tuning
 defined • 3-4

U

UIC (user identification code) • 1-13
UIC-based protection • 4-27
UP command • 10-15
UPDATE_IF attribute • 8-10
UPDATE attribute
 of ACCESS primary • 7-4
 of SHARING primary • 7-5
Update-if option • 8-5
Update operation • 3-11
Update service • 8-2, 8-5 to 8-6
 high-level language equivalents • 8-2
 run-time options • 9-24 to 9-25
USER_FILE_OPEN attribute • 7-5
USER_INTERLOCK attribute • 7-5, 7-9
User buffer
 address • 9-21
 size • 9-21
User identification code
 See UIC

V

Variable format with fixed-length control
 See VFC
Variable-length record
 with D format • 2-11
 with V format • 2-11
VAX BASIC
 USEROPEN routine • 5-13, 9-5
VAXcluster • 3-34
 locking considerations • 3-35
VAX MACRO • 3-14, 3-18, 3-33, 4-3

VAX MACRO (cont'd.)
 and VAX RMS • 9-5
 VAX RMS (Record Management Services) •
 1-14
 allocating buffers • 3-14, 3-18
 bucket splits • 3-27
 calculating extension size • 3-12
 calculating file extension size • 3-6
 connect-time options • 4-3
 control blocks • 1-15, 4-19
 creation-time options • 4-3, 4-22
 data structures • 1-15
 deferred-write operation • 3-18, 3-33
 in indexed files • 3-19
 MACRO parameter • 3-14
 placing file information in prologue • 3-19
 use of multiblocks • 3-13
 using with languages • 1-14
 with Prolog 3 files • 10-36
 VAX RMS (Record Management Services)
 utilities
 ANALYZE/RMS_FILE • 1-16
 CONVERT • 1-17
 CONVERT/RECLAIM • 1-17
 CREATE/FDL • 1-18
 EDIT/FDL • 1-19
 VAX RMS option
 selection • 9-1
 VFC • 2-13, 3-11, 3-12
 VFC record format • 1-4
 VIEW command • 4-5
 Virtual block position option • 4-40
 Volume
 defined • 1-6
 multidisk • 3-27
 positioning • 3-27
 VOLUME attribute • 4-40
 Volume number option • 4-40
 Volume set
 defined • 1-8

W

WAIT_FOR_RECORD attribute • 7-14
 Wait service • 8-7

Wait service (cont'd.)
 and asynchronous operations • 8-23
 Wildcard character
 See also File specification
 and multiple file locations • 5-10
 program preprocessing • 5-10 to 5-18
 Window • 9-9 to 9-11
 Window size • 10-35
 Working set • 1-20
 default • 1-20
 extent • 1-20
 quota • 1-20
 WSDEFAULT system parameter • 1-20
 WSEXTENT system parameter • 1-20
 WSQUOTA system parameter • 1-20

X

XAB (extended attribute block) • 1-15, 4-2
 date and time fields • 4-36
 key definition fields • 4-37
 protection fields • 4-37
 XAB\$_AID field • 4-39
 XAB\$_ALN field
 XAB\$_CYL option • 4-40
 XAB\$_RFI option • 4-40
 XAB\$_VBN option • 4-40
 XAB\$_AOP field
 XAB\$_HRD option • 4-40
 XAB\$_ONC option • 4-39
 XAB\$_BKZ field • 3-29, 4-36, 7-23, 7-24
 XAB\$_ALQ field • 4-39
 XAB\$_AOP field
 XAB\$_CBT option • 4-39
 XAB\$_CTG option • 4-39
 XAB\$_LOC field • 4-40
 XAB\$_DEQ field • 4-40
 XAB\$_VOL field • 4-40

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital



No Postage
Necessary
if Mailed in the
United States



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698



Do Not Tear - Fold Here

Cut Along Dotted Line

READER'S COMMENTS

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of user/reader that you most nearly represent:

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) _____

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code _____
or Country

Do Not Tear - Fold Here and Tape

digital



No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SSG PUBLICATIONS ZK1-3/J35
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, NEW HAMPSHIRE 03062-2698



Do Not Tear - Fold Here

Cut Along Dotted Line

NOTES

NOTES

